UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

Ciências
ULisboa

Byzantine State Machine Replication for the Masses

Doutoramento em Informática
Especialidade Ciência da Computação

João Catarino de Sousa

Tese orientada por:
Prof. Doutor Alysson Neves Bessani

Documento especialmente elaborado para a obtenção do grau de doutor

2017

UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

Ciências
ULisboa

Byzantine State Machine Replication for the Masses

Doutoramento em Informática
Especialidade Ciência da Computação

João Catarino de Sousa

Tese orientada por:
Professor Doutor Alysson Neves Bessani

Júri:
Presidente:

- Doutor Nuno Fuentecilla Maia Ferreira Neves, Professor Catedrático da Faculdade de Ciências da Universidade de Lisboa

Vogais:

- Doutor Christian Cachin, *Researcher*
  IBM Research-Zurich, na qualidade de individualidade de reconhecida competência (Suíça)
- Doutor Rolando da Silva Martins, Professor Auxiliar Convidado
  Faculdade de Ciências da Universidade do Porto
- Doutor Nuno Manuel Ribeiro Preguiça, Professor Associado
  Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa
- Doutor Luís Manuel Pinto da Rocha Afonso Carriço, Professor Catedrático
  Faculdade de Ciências da Universidade de Lisboa
- Doutor Alysson Neves Bessani, Professor Associado
  Faculdade de Ciências da Universidade de Lisboa (orientador)

Documento especialmente elaborado para a obtenção do grau de doutor

Doutoramento financiado pela Fundação para a Ciência e a Tecnologia
SFRH / BD / 70736 / 2010

2017

Resumo

A técnica de replicação máquina de estados é um paradigma popular usado em vários sistemas distribuídos modernos. No entanto, apesar da adoção deste paradigma em sistemas reais tolerantes a faltas por paragem, ainda existem poucos exemplos de sistemas reais tolerantes a faltas bizantinas. Segundo a nossa experiência nesta área de investigação, isto deve-se ao fato de existirem poucas concretizações robustas para replicação máquina de estados tolerante a faltas bizantinas, assim como uma perda de desempenho demasiado elevada, especialmente em ambientes geo-replicados. A razão fundamental para a existência destes obstáculos vem dos protocolos distribuídos necessários para assegurar este tipo de resiliência. Esta tese tem como objetivo explorar metodologias para a robustez e eficiência da replicação máquina de estados.

A primeira contribuição da tese é o algoritmo Mod-SMaRt, um protocolo modular que preserva latência ótima em termos de passos de comunicação executados pelos processos. Sendo um protocolo modular, torna-se mais simples de validar e concretizar, o que resulta em maior robustez; ao preservar troca de mensagens ótima entre processos, também é capaz de entregar um desempenho desejável.

A segunda contribuição consiste em concretizar o protocolo Mod-SMaRt na ferramenta BFT-SMART, uma biblioteca fiável de alto desempenho, mantida e melhorada ao longo de todo o período correspondente ao doutoramento, capaz de suportar arquiteturas multi-núcleo, reconfiguração do grupo de réplicas, e uma API de programação flexível.

A terceira contribuição consiste em um protocolo derivado do Mod-SMaRt designado WHEAT, que usa otimizações que demostraram serem eficientes na redução da latência segundo uma avaliação prática em ambiente geo-replicado.

Adicionalmente, foram também realizadas avaliações de ambos os protocolos quando aplicados num middleware para base de dados relacionais, e num serviço de ordenação para uma plataforma blockchain. Ambas as avaliações revelam

resultados encorajadores para ambos os sistemas e validam o trabalho realizado em contexto geo-distribuído.

Palavras Chave: sistemas distribuídos, tolerância a faltas bizantinas, replicação máquina de estados, geo-replicação, difusão atómica, consenso

Abstract

The state machine replication technique is a popular approach for building Byzantine fault-tolerant services. However, despite the widespread adoption of this paradigm for crash fault-tolerant systems, there are still few examples of this paradigm for real Byzantine fault-tolerant systems. Our view of this situation is that there is a lack of robust implementations of Byzantine fault-tolerant state machine replication middleware, and that the performance penalty is too high, specially for geo-replication. These hindrances are tightly coupled to the distributed protocols used for enforcing such resilience. This thesis has the objective of finding methodologies for enhancing robustness and performance of state machine replication systems.

The first contribution is Mod-SMaRt, a modular protocol that preserves optimal latency in terms of the communications steps exchanged among processes. By being a modular protocol, it becomes simpler to validate and implement, thus resulting in greater robustness; by also preserving optimal message-exchanges among processes, the protocol is capable of delivering desirable performance.

The second contribution is concerned with implementing Mod-SMaRt into BFT-SMART, a reliable and high-performance codebase that was maintained and improved over the entire course of the PhD that offers multicore-awareness, reconfiguration support, and a flexible API.

The third contribution presents WHEAT, a protocol derived from Mod-SMaRt that uses optimizations shown to be effective in reducing latency via a practical evaluation conducted in a geo-distributed environment.

We additionally conducted an evaluation of both BFT-SMART and WHEAT applied to a relational database middleware and an ordering service for a permissioned blockchain platform. These evaluations revealed encouraging results for both systems and validated our work conducted in the geo-distributed context.

Keywords: distributed systems, Byzantine fault tolerance, state machine replication, geo-replication, total order broadcast, consensus

Resumo Estendido

A técnica de replicação máquina de estados é um paradigma popular usado em vários sistemas distribuídos modernos. A ideia geral consiste em suportar um número arbitrário de clientes que enviam operações para um grupo de réplicas (máquinas de estado) que executam a mesma sequência de operações para um serviço distribuído, apesar da ocorrência de faltas numa fração das ditas réplicas. Este paradigma é também a principal abordagem para concretizar tolerância a faltas bizantinas – uma extensão da tolerância a faltas por paragem, onde se assume que processos falham ao demonstrar qualquer tipo de comportamento diferente do esperado, em vez de simplesmente pararem a sua execução.

No entanto, apesar da adoção deste paradigma em sistemas reais tolerantes a faltas por paragem, ainda existem poucos exemplos de sistemas reais tolerantes a faltas bizantinas. Segundo a nossa experiência nesta área de investigação, existem dois obstáculos que impedem a adoção deste paradigma para o modelo de faltas bizantino: (1) poucas concretizações robustas de middleware para replicação máquina de estados tolerante a faltas bizantinas (isto é, maioritariamente protótipos criados para validar inovações em artigos científicos), e (2) perda de desempenho demasiado elevada, especialmente em ambientes geo-replicados. A razão fundamental para a existência destes obstáculos vem dos protocolos distribuídos necessários para assegurar que todas as réplicas obtêm a sequência de operações mencionada anteriormente. Protocolos congeminados para tolerar faltas bizantinas – que normalmente requerem vários passos de comunicação entre réplicas com elevada quantidade de troca de mensagens – são significativamente mais complicados e exibem maior latência que as suas variantes para faltas por paragem, especialmente para geo-replicação.

Esta tese encontra-se inserida dentro da área de investigação sobre tolerância a faltas bizantinas com foco na técnica de replicação máquina de estados, isto é, propõe metodologias e ferramentas para a construção de sistemas que usam este paradigma. O objetivo é investigar técnicas que facilitem a concretização

de sistema tolerantes a faltas bizantinas robustos e eficientes. Como tal, foram investigadas estratégias para permitir que a técnica de replicação máquina de estados seja concretizada da forma mais eficiente e sucinta possível. Para além disso, foram também investigadas técnicas para otimizar este tipo de replicação em ambientes geo-replicados.

A primeira contribuição da tese é realizada ao nível teórico. Um requisito fundamental de qualquer protocolo de replicação máquina de estado consiste em garantir que todas as operações dos clientes chegam às réplicas pela mesma ordem. Este comportamento requer o uso de um algoritmo de ordenação total, que se sabe ser equivalente a resolver um consenso distribuído. Esta classe de algoritmos é reconhecida como sendo bastante complicada, e por isso torna-se difícil de obter uma concretização robusta e correta. A maneira como abordámos este problema consistiu em procurar uma forma de obter um protocolo de replicação que fosse simultaneamente modular e ótimo em termos de passos de comunicação. Se o protocolo for modular, também é mais simples de validar e concretizar, possibilitando por isso melhor robustez; se para além disso também for um protocolo ótimo em termos de passos de comunicação, temos maior eficiência.

O resultado desta investigação é o protocolo Mod-SMaRt, uma transformação de um algoritmo de consenso bizantino para um protocolo de replicação máquina de estados construído diretamente por cima de uma primitiva de consenso aumentada. Tal primitiva permite que o protocolo se mantenha ótimo em termos de passos de comunicação sem quebrar a desejada modularidade. Isto é possível através do uso do *VP-Consensus*, uma primitiva de consenso aumentada que pode ser obtida de algoritmos de consensos pré-existentes através de simples modificações. Esta primitiva é composta por propriedades adicionais que impõem as seguintes restrições: o *input* submetido à primitiva tem que satisfazer um predicado e o seu *output* tem de incluir uma prova criptográfica que vincula esse *output* à instância de consenso onde foi obtido.

A segunda e principal contribuição da tese é realizada ao nível prático. Como mencionado anteriormente, existem poucas soluções para replicação máquina de estados bizantina apesar de várias publicações científicas. A razão fundamental

para essa discrepância deve-se ao facto de a esmagadora maioria dos trabalhos realizados na área oferecerem concretizações que funcionam como provas de conceito usadas para uma avaliação prática, mas que raramente são mantidas a longo prazo. Esta observação motivou a concretização de uma ferramenta que foi mantida durante todo o período do doutoramento. O resultado deste esforço é a biblioteca BFT-SMaRt, uma biblioteca Java de código aberto que concretiza de forma eficiente e robusta o algoritmo Mod-SMaRt e a primitiva *VP-Consensus* mencionada anteriormente. Para além disso, a biblioteca suporta também mecanismos para transferência de estado entre réplicas que recuperam de faltas, adição/remoção de réplicas durante a execução do sistema, suporte para arquiteturas multi-núcleo, e uma API de programação flexível.

A terceira contribuição consiste em investigar técnicas de otimização de latência em protocolos de replicação para ambientes geo-distribuídos e aplicá-las ao trabalho desenvolvido previamente. Como tal, foi realizada uma avaliação prática de algumas otimizações para protocolos de replicação máquina de estados propostas na literatura, concretizando-as na biblioteca BFT-SMaRt. Os resultados mostraram que enquanto algumas dessas otimizações são muito eficientes, outras não trazem benefícios significativos. As conclusões tiradas dessa avaliação influenciaram a criação do WHEAT, um protocolo de replicação máquina de estado para ambientes geo-replicados que usa as otimizações que demonstraram ser mais eficientes a reduzir latência neste tipo de ambientes. A principal inovação do WHEAT em relação a outros protocolos da literatura consiste em dois esquemas de distribuição de votos que permitem melhor desempenho dentro de ambiente heterogéneos através do uso de réplicas adicionais, mas sem violar a correção do protocolo.

A última contribuição consiste em introduzir os protocolos desenvolvidos em protótipos representativos de sistemas geo-replicados reais, assim como uma avaliação prática para cada sistema. Os sistemas escolhidos foram um *middleware* para bases de dados relacionais baseado no protocolo Byzantium e um serviço de ordenação para a plataforma de blockchain Hyperledger Fabric. Os resultados das avaliações demonstram que ambos os sistemas exibem desem-

penho aceitável e validam o trabalho da tese realizado num contexto de geo-replicação.

Acknowledgements

I genuinely struggle to find what to say as I type this. Lets see if I can work it out and not forget anyone, because this thesis would simply not exist without the people mentioned below.

First and foremost, I want – and *need* – to thank my advisor, Professor Alysson Bessani. Professor Bessani insightful ideas, important conversations, crucial debates, diligent guidance, and heartfelt motivational rants were extremely precious during the whole journey. His tireless support was key to bring this thesis to a conclusion, and his uncompromising nature to push only towards the absolute best I could do was fundamental for this thesis to be what it is.

I also want to thank all my past and present LaSIGE colleagues with whom I went to the cafetaria with. Those occasions also helped me a great deal, be it by being there to give me advice when I most needed, laugh at my silly outbursts, or allowing me to talk about Rammstein. I prefer not to say any names because I would run the risk of forgetting someone; this way anyone that reads these acknowledges knows that I am indeed including them. If you are my friend but never worked in LaSIGE, this acknowledgement applies to you too.

Very importantly, I want to thank my mother for her unlimited patience and support throughout all these years. This will be the shortest acknowledgement, not because I feel ungrateful, but because there are no combination of words that can properly express how important she is to me.

Finally, I gratefully acknowledge the financial support from European Comission (EC) through projects TCLOUDS (FP7/2007-2013, ICT-257243), SUPERCLOUD (643964) and from national funds through Fundação para a Ciência e a Tecnologia (FCT) with the projects LaSIGE (UID/ CEC/00408/2013), RCClouds (PTDC/EIA-EIA/115211/2009), IRCoC (PTDC/EEI-SCR/6970/2014), and the PhD scholarship granted to (SFRH / BD / 70736 / 2010).

Dedico esta tese à minha avó, que infelizmente não viveu anos suficientes para testemunhar esta aventura.

Contents

List of Figures

List of Tables

List of Algorithms

List of Publications

International Conferences

- João Sousa, Bruno Branco e Brito, Alysson Bessani, Marcelo Pasin, *Desempenho e Escalabilidade de uma Biblioteca de Replicação de Máquina de Estados Tolerante a Faltas Bizantinas*, in Proceedings of the 3rd Simpósio de Informática, Coimbra, Portugal, Sept. 2011.
- João Sousa, Alysson Bessani, *From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation*, in Proceedings of the 9th European Dependable Computing Conference, Sibiu, Romenia, May 2012.
- Vinicius Vielmo Cogo, André Nogueira, João Sousa, Marcelo Pasin, Hans P. Reiser, Alysson Bessani, *FITCH: Supporting Adaptive Replicated Services in the Cloud*, in Proceedings of the 13th IFIP International Conference on Distributed Applications and Interoperable Systems, Jim Dowling, Francois Taïani, Eds., Florence, Italy, Jun. 2013.
- Alysson Bessani, João Sousa, Eduardo Alchieri, *State Machine Replication for the Masses with* BFT-SMART, in Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Atlanta, USA, Jun. 2014.
- João Sousa, Alysson Bessani, *Separating the WHEAT from the Chaff: An Empirical Design for Geo-Replicated State Machines*, in Proceedings of the 34th Symposium on Reliable Distributed Systems, Montreal, Canada, Sept. 2015.

- Eduardo Alchieri, João Sousa, Alysson Bessani, *Especificação de Replicação Máquina de Estados Dinâmica*, in Proceedings of the SXXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, Salvador, Bahia, Brazil, May 2016 .
- João Sousa, Alysson Bessani, Marko Vukolić, *A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform*, in Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Luxembourg City, Luxembourg, Jun. 2018.

Workshop Papers and Fast Abstracts

- Alysson Bessani, João Sousa, Eduardo Alchieri, ... *And State Machine Replication for All with* BFT-SMART, Poster in the 7th ACM SIGOPS/EuroSys European Systems Conference, Apr. 2012.
- João Sousa, Alysson Bessani, *Evaluating State Machine Replication Over a WAN*, fast abstract in Workshop on Planetary-Scale Distributed Systems, Braga, Portugal, Sept. 2013.
- João Sousa, Alysson Bessani and Marko Vukolic, *A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform*, Short research statement in 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers, Las Vegas, Nevada, USA, Dec. 2017.

1 Introduction

In computer science, replication is the process of creating and managing duplicate versions of information. This is done by propagating data or computations among a set of processes known as replicas, which can either be co-located within the same data-center or scattered across the globe. This strategy is used to provide systems with enhanced performance, greater availability and support for fault tolerance. The two main approaches to replication are either primary-backup (Alsberg & Day, 1976; Budhiraja *et al.*, 1993) or state machine replication (Lamport, 1978; Schneider, 1990). When the priority is to create a system that offers high availability, primary-backup replication is typically the chosen approach. On the other hand, when the primary goal is to provide a greater degree of fault tolerance to critical services, state machine replication (SMR) is the more appropriate option. The basic idea is to have an arbitrary number of clients issuing operations to a set of replicas (state machines) that execute the same sequence of operations for a service despite the occurrence of faults on a fraction of these replicas. This technique is adopted by many modern distributed systems, ranging from cluster-based coordination services (e.g., Chubby (Burrows, 2006) and Zookeeper (Hunt *et al.*, 2010)), decentralized storage systems (e.g., Cassandra (Lakshman & Malik, 2010)[1] and Megastore (Baker *et al.*, 2011)), Platform-as-a-Service providers (e.g., Microsoft Autopilot (Isard, 2007), RethinkDB (Walsh *et al.*, 2009)[2] and Amazon Web-Services (Elisha & Hamilton, 2014)), replicated key-value stores (e.g., JSimpleDB (Cobbs,

[1] Cassandra implements SMR for its lightweight transactions since 2013
[2] RethinkDB implements SMR for automatic failover since 2015

2016) and ETCD[1]), SDN controllers (e.g., OpenDayLight[2]) and transactional database systems (e.g., Spanner (Corbett et. al, 2013) and TiDB[3]).

SMR is also the main technique for implementing Byzantine Fault Tolerance (BFT). This is a super-set of standard Crash Fault Tolerance (CFT) where processes are expected to fail by displaying any type of deviation from their specified behavior, rather than simply failing by stopping to execute (Lamport *et al.*, 1982). In fact, the last two decades have seen a significant amount of papers on BFT SMR protocols (e.g., Abd-El-Malek *et al.* (2005); Aublin *et al.* (2015); Behl *et al.* (2017); Castro & Liskov (2002); Cowling *et al.* (2006); Kapitza *et al.* (2012); Kotla *et al.* (2009); Liu *et al.* (2016); Veronese *et al.* (2013)). However, despite the widespread adoption of SMR for standard crash faults in production systems, there are still very few examples of practical deployment of this technique for Byzantine faults. Only fairly recently are organizations seeking to adopt this paradigm – primarily due to the advent of blockchain technology (Furlonger & Valdes, 2016), where coping with malicious behavior is of high importance to preserve the integrity of transactions associated with crypto-currency and smart-contracts (Cachin, 2016; Cachin & Vukolic, 2017b; Kwon, 2016; Martino, 2016). Our view of this situation is that there are two key obstacles to the widespread adoption of this paradigm: (1) a lack of robust implementations of BFT SMR middleware (i.e., mostly prototypes used for validating novel ideas in papers), and (2) the performance penalty associated with this approach is too high, specially if the system is intended to be geo-replicated, such as the case of blockchain platforms. The fundamental reason why these hindrances exists stems from the core of a BFT replicated state machine deployment: the protocol that is used for establishing the aforementioned sequence of operations across replicas. These BFT protocols – which usually requires multiple communication steps with high number of message exchange – are not only notoriously more complicated than their CFT counterparts, but display higher latency overhead, specially over geo-distributed environments. Hence, the general perception is that implementing BFT SMR is a complex and inefficient approach to fault tolerance.

[1] https://coreos.com/etcd
[2] https://www.opendaylight.org/
[3] https://pingcap.com/docs

1.1 Objectives & Contributions

The thesis was developed within the Navigators group of the Large-Scale Informatics Systems Laboratory (LASIGE) at the Informatics Department of the Faculty of Science of the University of Lisbon. It is inserted within the context of Byzantine fault tolerance with a focus on SMR, i.e., it proposes methodologies and tools that contribute to build systems that use such paradigm. The objective is related with the investigation of techniques that facilitate the implementation of BFT services. The thesis has therefore two main objectives: *robustness* and *performance* of SMR.

The first objective is about identifying a strategy that allows the state machine approach to be implemented and extended in the most straight-forward way possible, by providing a modular replication framework and a reliable codebase from which services can be developed from. The second objective is about ensuring these services can still exhibit an acceptable performance in spite of using a replication approach that has a reputation of being inefficient and unpractical in the industry. In particular, this thesis work includes an effort to render this type of replication practical in geo-distributed environments. Taking this into consideration, the main topics of research this thesis contributes to are as follows:

State Machine Replication Protocols

Our first step towards a robust and efficient BFT state machine solution starts at the theoretical level. A fundamental requirement of any SMR protocol is to enforce all requests to be delivered across replicas by the same order. Such ordering requires a total order broadcast algorithm, which is known to be equivalent to solving the consensus problem (Correia *et al.*, 2006; Hadzilacos & Toueg, 1993; Milosevic *et al.*, 2011). This class of algorithms are well known to be significantly complex, which in turn makes it hard to achieve a robust and correct implementation (Chandra *et al.*, 2007). Our general approach to this problem consisted in investigating a way to obtain a SMR protocol that is both modular and optimal in terms of communication steps. If the protocol is modular, it becomes simpler to validate and implement, thus resulting in greater robustness; if it enables optimal message-exchanges across processes, it is also capable of delivering desirable performance. Moreover, it is also important that the formalization of such protocol is made as clear as possible to facilitate implementation.

Our main contribution to this topic is Mod-SMaRt, a new transformation from Byzantine consensus to BFT SMR built directly around an augmented consensus abstraction. The usage of such abstraction enables the algorithm to remain modular while still preserving optimal latency in terms of communications steps. It is the first optimal transformation from consensus to SMR. This is achieved by using VP-Consensus, an augmented consensus abstraction which can be obtained from pre-existing consensus algorithms by making simple modifications. This abstraction is defined by additional properties that impose *constraints* on the proposals and *guarantees* on the obtained decisions. More precisely, proposed input must satisfy a given predicate and the output must include a cryptographic proof that binds that output to its respective consensus instance.

State Machine Replication Implementations

Our second step towards a robust and efficient BFT state machine solution is at the practical level. As mentioned previously, there are still scarce options for reliable BFT SMR deployments in spite of the several papers on the subject. The fundamental reason for this discrepancy between research and engineering is due to the overwhelming majority of those works providing a proof-of-concept codebase that is adequate to conduct their experimental evaluation, but which is rarely maintained over the long term. Moreover, the lack of robust implementations also becomes an hindrance to its own research field, given that authors needed to either create their own codebase from scratch to evaluate their prototypes, or work on unstable codebases. This lack of options for BFT SMR implementations is tightly coupled to the fact that these types of systems are notoriously hard to implement, even within the scope of CFT (Beyer *et al.*, 2016; Chandra *et al.*, 2007). This has already motivated the design of more understandable algorithms such as Raft (Ongaro & Ousterhout, 2014) – which has been quickly adopted in production systems (e.g., Walsh *et al.* (2009), Cobbs (2016)) – but so far the same effort has not been devoted to BFT systems. This observation lead us to the core effort of the thesis, which is concerned with taking the aforementioned modular algorithm and implementing it into a codebase that was maintained and improved over the entire course of this PhD. By devoting time and effort to such codebase, we aimed at creating a reliable implementation of the SMR technique that can not only demonstrate

that it is possible to achieve acceptable performance with this type of replication, but also serve as a building block to future research done in the field.

Our main contribution for this topic – as well as the flagship contribution developed within the PhD – is BFT-SMART, an open-source Java library implementing the aforementioned Mod-SMaRt transformation and VP-Consensus abstraction. Some of the key features of BFT-SMART that distinguishes it from similar works are improved reliability, modularity as a first-class property, multicore-awareness, reconfiguration support, and a flexible API. The effort expended at developing and maintaining the library over the course of the PhD resulted in a stable codebase that is used by many authors to develop their research in the field of Byzantine fault tolerance (e.g., Behl *et al.* (2015); Kapitza *et al.* (2012); Martins *et al.* (2013); Porto *et al.* (2015)), be it by using the library as a building block or extending it to produce new prototypes of innovative systems. Furthermore, the system is also being used in some blockchain platforms such as Corda[1] and Symbiont Assembly[2] as a BFT consensus implementation.

Geo-Replication

The third topic of the thesis consists in exploring ways to optimize our work for a geo-replicated setting. The primary motivation behind the academic research on geo-replication is to render critical services able to survive disasters. However, aside from systems such as Megastore (Baker *et al.*, 2011) or Spanner (Corbett et. al, 2013), there are still few examples of practical deployments of the SMR technique within a wide-area context. The vast majority of productions systems that can be used to create resilient critical services through geo-replication – such as Amazon's DynamoDB (DeCandia *et al.*, 2007), CouchDB (Anderson *et al.*, 2010), and Memcached (Fitzpatrick, 2004; Nishtala *et al.*, 2013) – provide weaker consistency semantics than that of SMR in order to satisfy end-users with highly-available services (Bailis *et al.*, 2013; Saito & Shapiro, 2005). Because of the necessity of providing high-availability even in a wide-area scenario, it is important to explore ways to optimize the work developed in this thesis into this type of environment.

The work developed for this topic began with a practical evaluation of some representative optimizations proposed in the literature for state machine protocols, by implementing

[1] https://www.corda.net/2017/03/corda-m9-1-released/
[2] https://symbiont.io/technology/assembly/

them on BFT-SMART and running the experiments in geo-distributed environments. Interestingly, the results show that some optimizations for improving the latency of geo-replicated state machines do not bring significant benefits, while others were shown to be very effective. The results obtained from this evaluation guided the design of WHEAT, a new configurable crash and Byzantine fault-tolerant SMR protocol derived from Mod-SMaRt that uses the optimizations that were shown as most effective in reducing latency in a geo-distributed scenario. One of the key features that distinguishes WHEAT from most protocols proposed in the literature are two novel vote assignment schemes designed to preserve protocol correctness while also allowing for performance improvement within heterogeneous environments such as wide-area networks.

Services

The final topic of this thesis revolves around applying our SMR protocols and systems into prototypes that represent practical replicated databases. This is motivated not only for the necessity to render critical infrastructures able to survive disasters, but also to explore alternative approaches for devising geo-replicaded systems. Such approaches are meant to facilitate the engineering of new systems that provide more functionality while embracing weaker assumptions about its environment. As an example of this, consider the Spanner transactional database (Corbett et. al, 2013). This geo-replicated system is able to exhibit acceptable performance for end users, but it does not support full-fledged relational tables and requires an API that uses GPS and atomic clocks to infer timeliness uncertainty – not to mention that it also only withstands crash faults. We would like to find new strategies that mitigate this kind of limitations.

Another strong motivation for this final topic is linked to the very recent emergence of blockchain platforms (Furlonger & Valdes, 2016). These platforms are comprised by a peer-to-peer network with hundreds of geographically-dispersed nodes that maintained a distributed ledger of transactions (Buterin, 2015; Nakamoto, 2009). Because of the critical nature of crypto-currencies and smart-contracts, these platforms need to account for malicious behavior of a subset of those nodes - which is something that BFT SMR is capable of withstanding (Vukolić, 2015, 2017). This reinvigorated interest in Byzantine fault tolerance also led us to apply the work developed in the context of geo-replication into one such

blockchain platforms.

Our final contribution is a practical evaluation of BFT-SMART and WHEAT applied to two types of geo-replicated systems: a relational database middleware based on the Byzantium (Garcia *et al.*, 2011), and an ordering service for the Hyperledger Fabric blockchain platform (Cachin, 2016). In the case of Byzantium, our work also serves to (1) propose a geo-replicated system that does not carry limitations akin to Spanner's; and (2) complements the original work with an wide-area evaluation. In the case of Hyperledger Fabric, the contribution is extended by providing an architecture and implementation for the aforementioned ordering service, as well as presenting a proof-of-concept that demonstrates the potential of this type of systems.

1.2 Thesis Structure

The remaining of this thesis is organized as follows:

Chapter 2: Background. This chapter provide the context for the thesis and presents the related work. More precisely, it unpacks some basic concepts widely adopted in Byzantine fault tolerance – such as the consensus problem and related challenges – and provides a brief historical retrospective of the emergence of this research area. Many relevant works related to SMR are addressed and discussed, including efforts that bring this technique into geo-distributed scenarios.

Chapter 3: Mod-SMaRt. This chapter formally describes the VP-Consensus abstraction. Following this, we describe how to use the abstraction to obtain the Mod-SMaRt protocol. Possible optimizations to Mod-SMaRt are also discussed at the end of the chapter. Appendix A and B complement this chapter by presenting the proof of correctness for Mod-SMaRt and describing how to obtain the VP-Consensus out of a pre-existing consensus algorithm, respectively.

Chapter 4: BFT-SMART. Following the work developed in Chapter 3, we present the BFT-SMART replication library, the principal contribution developed during the course of this PhD. We discuss the library's architecture and API, as well as the lessons learned from the practical effort geared towards maintaining the codebase. We also include a thorough

experimental evaluation of the library, exploring how its performance evolves upon fiddling with the size of the requests/replies, the number of replicas present in the system, ratio of read/write operations, and the number of cores available per machine. We also showcase how BFT-SMaRt's performance fares against other BFT prototypes, and we find that the library outperforms all the chosen prototypes.

Chapter 5: WHEAT. By re-purposing the codebase described in Chapter 4, we present a study of protocol optimizations conducted on geo-distributed settings and describe how the results guided the design of WHEAT. The chapter also includes a description of the vote assignment schemes used by this modified version of Mod-SMaRt and an experimental evaluation of WHEAT, where the protocol exhibits a significant improvement over the standard original one. Appendix C complements this chapter by presenting the proof of correctness for the voting schemes used by the WHEAT protocol.

Chapter 6: Applications. This chapter describes the integration of WHEAT into of a geo-replicated transactional middleware for relational databases and an ordering service used by a blockchain platform. Both of these systems are given a practical evaluation within a local cluster and in a wide-area network. We present measurements using BFT-SMaRt and WHEAT for both the local and wide-area settings.

Chapter 7: Conclusions. This chapter concludes the thesis by discussing its impact on the research area and its adoption in some blockchain platforms, as well as proposing some future work.

2 Background

This chapter discusses the body of research on which this thesis is based on, in particular Byzantine fault tolerance and state machine replication. An overview of Byzantine fault tolerance is given is Section 2.1. The section also unpacks the terminology, system assumptions typically adopted in the research area, and a description of quorum systems. Section 2.2 presents the notion of consensus and state machine replication, which are the main topics of this thesis. Section 2.3 explains the emergence of Byzantine fault tolerance as a research area and describes the state of the art. Section 2.4 describes state machine replication solutions aimed for wide-area networks.

2.1 Overview

In distributed systems, fault tolerance (Randell *et al.*, 1978) is the body of techniques which enable a system to endure execution upon failures (possibly at the cost of reduced performance), rather than halting or deviating from its specified behavior. The system as a whole is not stopped due to problems either in hardware or software. The goal of fault tolerance is not to avoid faults, but rather to create systems that can withstand them. Fault-tolerant systems are typically based on the concept of redundancy. This usually demands that the system be partially or totally replicated, usually across multiple hosts connect by a network – hence, the importance of state machine replication (SMR) in this research area.

Byzantine fault tolerance is a sub-field of fault tolerance research within distributed systems. In classical fault tolerance, processes are assumed to fail only by stopping to execute.

On the other hand, in the *Byzantine faults model* (Lamport *et al.*, 1982), processes of a distributed system are allowed to fail in a arbitrary way, i.e., a fault is characterized as any deviation from the specified algorithm/protocol imposed on a process. Thus, Byzantine fault tolerance (BFT) is the body of techniques that aims at devising protocols, algorithms and services that are able to cope with such arbitrary behavior of the processes that comprise the system. Additionally, in Byzantine fault tolerance it is common practice to make assumptions as weak as possible, not only from the processes that comprise the system, but also from the network that connects them.

2.1.1 Terminology and Assumptions

One of the fundamental concepts in computer science is the notion of a *process*. In the context of distributed systems, a process is an instantiation of an algorithm that is being executed. A process is considered to be *correct* if it always complies to the execution specified by the algorithm that is instantiated. Otherwise, a process is deemed to be *faulty*. Furthermore, every process is assumed to have access to a bidirectional link to any other process in the system. Typically these links are modeled as being *fair-loss* (Lynch, 1996), an abstraction that assumes that each message can be lost an unbounded number of times. Nonetheless, if both sender and recipient are correct, the message is eventually delivered by the recipient, as long as the sender keeps re-transmitting it. This abstraction can be used to derive a more powerful abstraction called *perfect point-to-point links*, where messages are eventually delivered to the recipient, as long as both sender and recipient remain correct. If this abstraction is augmented with cryptographic functions that enable the recipient to verify if the message was indeed produced by the sender and was not fabricated by a malicious third-party, the systems provides *authenticated perfect point-to-point links*.

Byzantine faults can be devided in two types: (1) omission faults (e.g., crash faults, failing to receive a message, or failing to send one) and (2) commission faults (e.g., processing a request incorrectly, corrupting local state, and/or sending an incorrect/inconsistent reply to a request).

Another important concept in distributed systems is the notion of network synchrony and timeliness. Distributed algorithms created to execute across distinct processes need to make assumptions about the network that connects them. In particular, the following models are commonly considered (Hadzilacos & Toueg, 1993):

- *Synchronous System*: The network is timely and processes send and receive messages within known and fixed time bounds. Time bounds for local computations are also fixed and known;
- *Eventually Synchronous System*: Transmission/reception of messages is fixed but the bounds are unknown (or they are known but don't hold initially). The same principle applies to local computations;
- *Asynchronous System*: There are no guaranteed time bounds for neither transmission/reception of messages or local computations.

The assumption of a synchronous system is the least likely to hold up on real world settings. This can happen either due to periods where the system experiences heavy workload (e.g., under a distributed denial of service attack), or due the high variation of latency observed over the internet. Furthermore, protocols devised for the asynchronous model also offer the advantage of being more easily ported to various settings and environment, since they do not rely on the notion of time to fulfill their computations. On the other hand, some services cannot be built under the assumption of a completely asynchronous system model because their underlying algorithms require the notion of time. One such case is the SMR technique itself, as discussed in Section 2.2. Because of this, the model typically adopted in SMR is the eventually synchronous model, as it will be discussed in the following section in more detail.

Finally, another assumption that is commonly made about the system model is that the set of processes is finite and known *a-priori*. This is known as the *n-arrival model* and is widely common across the distributed algorithms literature. On the other hand, there are also distributed algorithms that that assume the set of processes to be infinite and changing over the system's lifespan. This is known as the *infinite arrival model* (Aguilera, 2004). These models are also usually referred to the *static* and *dynamic* system model, respectively.

2.1.2 Quorum Systems

The overwhelming majority of BFT literature relies on some form of quorum systems to enforce their safety properties. Given a set of hosts, a *quorum system* is a collection of sets of hosts (called *quorums*) such that any two quorums intersect by at least one common host (Garcia-Molina & Barbara, 1985; Gifford, 1979), as illustrated in Figure 2.1. Quorum

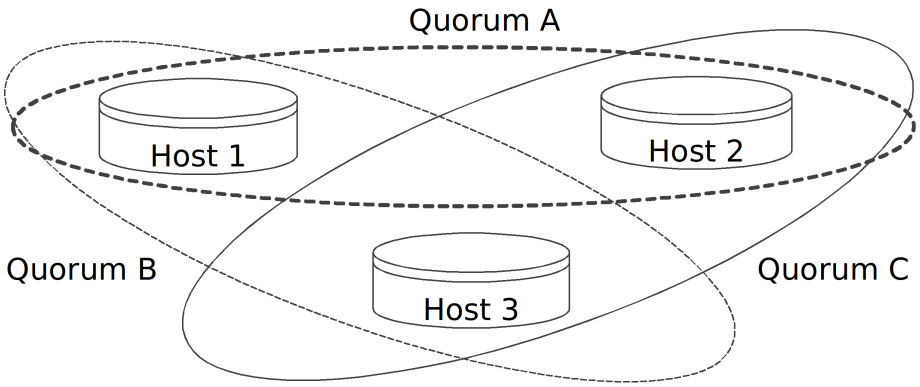


Figure 2.1: A quorum system comprised of 3 hosts. Quorum A contains hosts 1 and 2, quorum B contains hosts 1 and 3, and quorum C contains hosts 2 and 3.

systems are building blocks also used to implement a variety of services such as mutual exclusion (Agrawal & Abadi, 1991), distributed access control (Naor & Wool, 1998), and many protocols that must execute a distributed commit (Dolev *et al.*, 1982).

The most important guarantees that quorum-based protocols need to preserve are (1) all possible quorums overlap in some correct node, and (2) even with up to f failed replicas, there is always some quorum available in the system. In crash fault-tolerant protocols, quorums must overlap in at least one node. Such intersection is enforced by accessing a simple majority of nodes. More specifically, protocols access $\lceil \frac{n+1}{2} \rceil$ nodes out of $n \geq 2f + 1$. BFT protocols, on the other hand, usually employ disseminating Byzantine quorums (Malkhi & Reiter, 1998) with at least $f + 1$ replicas in the intersection. In this case, protocols access $\lceil \frac{n+f+1}{2} \rceil$ nodes out of $n \geq 3f + 1$.

2.2 Consensus and State Machine Replication

As mentioned in Chapter 1, there are two basic approaches to replication: *primary-backup* (Alsberg & Day, 1976; Budhiraja *et al.*, 1993) and *active replication* (Lamport, 1978; Schneider, 1990). Both assume the existence of clients, which issue commands to replicas (which are copies of the service). With primary-backup replication, there exists a primary replica, and all others are backups. Clients send their commands to the primary for execution. After the primary finishes processing the request (and before replying to the client), it updates the other backups with its state. On the other hand, in active replication each client

request is processed by all the replicas and all of them send a reply to the client. Active replication is more commonly referred to as *state machine replication* (SMR).

In the SMR model, an arbitrary number of client processes issue commands to a set of replica processes. These replicas implement a stateful service that changes its state after processing client commands, and sends replies to the clients that issued them. The goal of this technique is to make the state at each replica evolve in a consistent way, thus making the service completely and accurately replicated at each replica. In order to achieve this behavior, it is necessary to satisfy the following properties:

1. Operations from correct clients get executed;
2. If any two correct replicas r and r' apply operation o to state s, both r and r' will obtain state s';
3. Any two correct replicas r and r' start with state s_0;
4. Any two correct replicas r and r' execute the same sequence of operations $o_0, ..., o_i$.

The first three requirements can be easily fulfilled, but the last one requires a *total order broadcast* primitive (Hadzilacos & Toueg, 1993), which is equivalent to solving the *consensus problem* – one of the most studied problems in the field of distributed systems (Cachin, 2009; Lampson, 2001; Martin & Alvisi, 2006; Rütti *et al.*, 2010). Within this context, consensus aims at providing the following behavior: given some set of processes connected through some communication medium, each process will *propose* a value to be chosen across all processes in the set. Following this, all processes must *decide* exactly the same value. Such value must have been previously proposed by at least one of the processes on the set. More formally, protocols that solve the consensus problem typically satisfy the following properties (Hadzilacos & Toueg, 1993):

- *Termination*: Every correct process eventually decides exactly one value;
- *Agreement*: If a correct process decides v, then all correct processes eventually decide v;
- *Integrity*: If a correct process decides v, then v was previously proposed by some process.

Among all the research related to solving consensus under several fault and system models, there is one important result presented by Fischer *et al.* (1985): it is impossible to solve consensus under the asynchronous system model if at least one of the involved processes can fail by crash. This is usually referred to as the *FLP impossibility*, and implies that, any protocol that solves consensus, must take into consideration the timeliness of the network and of the other processes involved — meaning that the asynchronous system model needs to be expanded in same way.

Dwork *et al.* (1988) introduced the concept of partial synchrony mentioned in Section 2.1.1. This is a stronger model than the asynchronous one, in the sense that it provides a fixed upper bound for the computation time within processes and the delays caused by message transmission. However, such bounds are either not known *a priori*, or they shall only hold after an unknown instant in time, dubbed the *Global Stabilization Time*. In this approach, protocols must be built taking (eventual) timeliness explicitly into consideration.

Chandra & Toueg (1996) argued that the asynchronous model could be expanded using external failure detectors, allowing consensus to be solved in the presence of crash failures. This expanded model encapsulates the required timeliness within failures detectors — which are characterized as *oracles* — and allows for the construction of consensus protocols that do not need to take timeliness explicitly into consideration.

However, in spite of the aforementioned extensions to the asynchronous model, timeliness still remains an unavoidable requisite to devise any strongly consistent distributed service. In particular, a more general observation than the FLP result is stated by the CAP theorem (Gilbert & Lynch, 2002, 2012). This theorem takes into consideration the following dimensions: consistency, availability, and network-partitioning. The observation made is that there exists an unavoidable trade-off between these three properties: systems are unable to remain strongly consistent or available upon network partitions or the complete absence of synchrony. The network would need to be perfectly reliable and timely in order to fully and equally guarantee all the aforementioned dimensions. This is the main reason why most replicated system settle for eventual consistency (Saito & Shapiro, 2005) rather than strong consistency (Lamport, 1978).

Essentially, what both the FLP result and the CAP theorem show is the fact that there is a trade off between *safety* and *liveness*. Respectively, safety properties state what must never occur, whereas liveness properties state that, eventually, something benign shall occur. What is typically done in fault tolerance research, is to assume that safety properties are more

critical than liveness, and build protocols and algorithms that enforce safety regardless of the behavior of the network. Liveness is assumed to hold once the network displays desirable behavior (thus putting some onus of progress upon the network). Such behavior is assumed to occur *eventually*, during the system lifespan.

Many algorithms that implement SMR under the partial synchronous model have been proposed. The most well-known algorithm in the literature is Paxos (Lamport, 1998, 2001). Paxos assumes processes to take the following roles: clients (that issues operations), proposers (that propose ordering upon operations sent by clients), acceptors (which validate the proposers ordering), and learners (which learn the acceptors' decisions). Another well known state machine replication protocol is Viewstamped Replication (Oki & Liskov, 1988), which operates in similar way to Paxos, but makes no distinction between proposers, acceptors and learners. Both protocols are resilient to faulty processes by being quorum-based, thus requiring a minimum of $2f + 1$ processes in the system. Paxos is more widely adopted across the industry than Viewstamped Replication is, but it is also notoriously difficult to understand and implement (Chandra *et al.*, 2007). This drawback motivated the creation of a new SMR protocol named Raft (Ongaro & Ousterhout, 2014). Raft main design principle is understandability, which is achieved by using algorithm decomposition and by reducing the number of internal states each replica can transition to.

2.3 BFT State Machine Replication

Paxos, Viewstamped Replication, and Raft are prominent examples of protocols that implement SMR, but they assume processes fail only by crashing. On the other hand, there is wide and diverse research on SMR applied to the Byzantine fault model. This section provides an overview of the existing literature dedicated to this research area.

2.3.1 BFT Emergence

Early work assuming Byzantine faults suggested it to be too expensive to be of any widespread usage — either because it required many communication steps (which translates to increased latency), required expensive cryptographic computations (which is a bottleneck to the whole algorithm), or because it required synchrony for safety (Kihlstrom *et al.*, 2001; Reiter, 1994, 1995, 1996). Nonetheless, Castro and Liskov showed that state machine replication under

Byzantine faults is actually feasible, by presenting the Practical Byzantine Fault Tolerance (PBFT) protocol (Castro & Liskov, 1999, 2002). By constructing a replicated network file system (NFS) above PBFT, Castro and Liskov showed that their implementation could have a performance only 3% slower than a standard unreplicated NFS. The main difference between PBFT and previous proposals for BFT were that PBFT avoided expensive cryptographic operations such as digital signatures by using MAC vectors instead. Moreover, PBFT relied on synchrony only for liveness. PBFT requires disseminating Byzantine quorums to secure its safety properties, thus requiring a minimum of $3f + 1$ replicas in the system. This protocol spawned a *renaissance* in Byzantine fault tolerance research, and is considered the baseline for all BFT state machine replication protocols published afterwards.

2.3.2 PBFT-Derivated Protocols

One of the works which followed PBFT was Query/Update (Q/U) (Abd-El-Malek *et al.*, 2005), an optimistic quorum-based protocol that presents better throughput with larger number of replicas than other agreement-based protocols. However, given its optimistic nature, Q/U performs poorly under contention, and requires $5f + 1$ replicas. To overcome these drawbacks, Cowling *et al.* (2006) proposed HQ, a hybrid Byzantine fault-tolerant SMR protocol similar to Q/U in the absence of contention. However, unlike Q/U, HQ only requires $3f + 1$ replicas and relies on PBFT to resolve conflicts when contention among clients is detected.

Following Q/U and HQ, Kotla *et al.* (2009) proposed Zyzzyva, a speculative Byzantine fault-tolerant protocol, which was considered to be the fastest BFT protocol at the time. It is worth noticing that all these protocols tend to be more efficient than PBFT because they avoid the complete execution of a total order broadcast, relying on it only to solve corner cases.

Clement *et al.* (2009b) later showed that many of the aforementioned protocols are not robust enough even if a single malicious client is present; if such client is able to craft specially malicious requests, it can render the system's performance extremely low. To tackle this observation, the authors introduce a new protocol dubbed Aardvark, which introduces a pletora of pro-active mechanisms to prevent such malicious requests to make a negative impact on the performance. On the other hand, this gain in robustness comes at the cost of a slighly inferior performance — even in the absence of faults or attacks.

Clement *et al.* (2009a) also introduced the UpRight Cluster Service. This service consists of a Java library that uses a protocol named Zyzzyvark, which is derived from Zyzzyva but using the pro-active mechanisms introduced in Aardvark. Additionally, this library employs a separation of concerns at the following request processing stages of SMR: request quorum (used to generate a matrix of MACs equivalent to digital signatures (Aiyer *et al.*, 2008)), ordering, and execution (an idea first proposed by Yin *et al.* (2003), but only considering agreement and execution). Furthermore, UpRight's protocol assumes omission faults to be more common than Byzantine faults, and thus only a subset of all faults are expected to be Byzantine.

2.3.3 Protocols Resistant to Performance Degradation

All protocols mentioned so far rely on one designated replica — normally referred to as *leader*, *primary*, or *coordinator* — to assign an order to each valid request issued by clients. All other replicas must wait for the leader to make such assignment. Because of this, the performance of the entire system is dictated by the performance of the leader. However, if the leader becomes malicious, it might delay the order proposal to reduce global performance significantly — and yet, keep itself fast enough so it won't be suspected of being faulty (Amir *et al.*, 2011). As a countermeasure to this type of attack, the authors proposed Prime, an SMR protocol fitted with mechanisms for detecting this type of attack and blacklisting such malicious replica. However, the protocol requires knowledge about the networks bandwidth in order to accurately detect this behaviour.

Veronese *et al.* (2009) proposed a different mechanism to assess this problem in the Spinning protocol. Whereas in Prime (and other protocols) the system elects a new leader only when it is suspected of being faulty, Spinning periodically elects a new leader in a round-robin fashion, thus forcing all replicas to be elected as the leader during each iteration of the protocol. If any replica is suspected of being faulty, it is added to a blacklist, and not allowed to become the leader for some number of iterations.

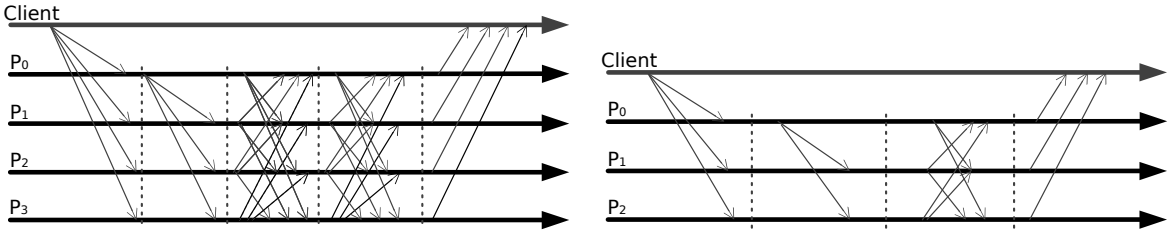
Aublin *et al.* (2013) proposed an alternative approach to (Amir *et al.*, 2011) for coping with performance-degrading leaders named Redundant-BFT (RBFT). Instead of devising a novel BFT SMR protocol with awareness of this type of attack, the authors devised a framework that uses $f + 1$ instances of any single SMR protocol picked from the literature, but with a different leader replica attached to each instance. All instances participate in

ordering requests, but only the requests ordered by the master instance are executed. In addition, the performance of each instances is closely analyzed by a monitoring mechanism to determine if the master instance provides lower performance than the others, in which case it is suspected of being malicious and therefore replaced.

2.3.4 Hybrid Protocols

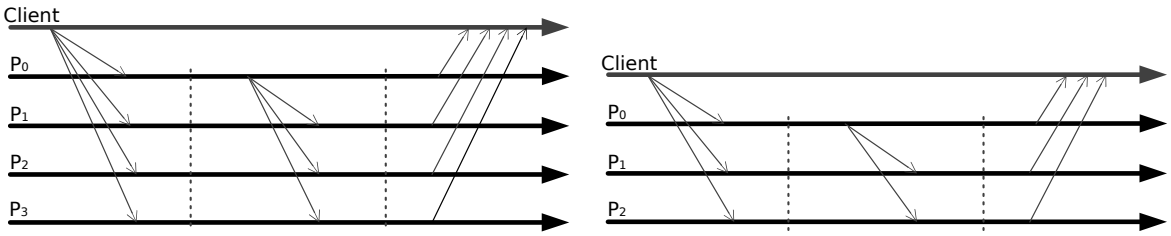
The fact that BFT SMR requires a theoretical minimum of $3f+1$ replicas spawned a research line aimed at reducing this minimum by means of an hybrid system model (Veríssimo *et al.*, 2003). The key idea is to assume the existence of a trusted and trustworthy sub-system that is shielded from arbitrary behavior by construction. An example of such BFT SMR protocols designed for an hybrid model are MinBFT and MinZyzzyvva, each one being derivations from Zyzzyvva and PBFT respectively (Veronese *et al.*, 2013). The protocols required a trusted component called *Unique Sequential Identifier Generator* (USIG), which defines an interface with operations to (1) increment a counter and generate a certificate that associates that value to a given message; and (2) verify if such certificate is correctly signed for that value/message pair. The simplicity of the interface enables to component to be realistically implemented using either virtualization or a trusted platform module (TPM) device. This component is assumed to deviate from its specified behavior only by crash. This allows the resulting protocols to use only $2f+1$ replicas – and one less communication steps in case of MinBFT – to implement state machine replication when compared to the original protocols. A comparison of PBFT's and MinBFT's (resp. Zyzzyva and MinZyzzyva) message patterns is illustrated in Figure 2.2 (resp. Figure 2.3). As it can be observed, PBFT (resp. Zyzzyva) requires 4 replicas, whereas MinBFT (resp. MinZyzzyva) needs 3 replicas to tolerate a single fault. In the particular case of MinBFT, an entire communication step can be omitted.

MinBFT was latter used as a building block for CheapBFT (Kapitza *et al.*, 2012), a BFT SMR protocol that uses a similar hybrid model, but that is capable of using only $f+1$ active replicas in the presence of network synchrony and absence of faulty behavior from replicas. During such period, the system executes a protocol that employs passive replication to propagate client operations, but is unable to make progress if it suspects any replica to be faulty. Because of this, CheapBFT executes a dedicated protocol which enables it to bring the inactive replicas up-to-date and switch to MinBFT, so that the system is able to make progress in the presence of f faults and a total of $2f+1$ replicas.



(a) PBFT message pattern. (b) MinBFT message pattern.

Figure 2.2: Comparison of PBFT and MinBFT message patterns, with client c sending an operation to the replicas.



(a) Zyzzyva message pattern. (b) MinZyzzyva message pattern.

Figure 2.3: Comparison of Zyzzyva and MinZyzzyva message patterns, with client c sending an operation to the replicas.

More recently, Behl *et al.* (2017) proposed Hybster, a BFT SMR protocol also for an hybrid model whose trusted component is based on Intel's Software Guard Extensions (SGX) technology. Previous protocols in this research line employed a sequential execution of the ordering protocol. The main advantage Hybster offers over CheapBFT or MinBFT is the fact that it allows for parallel execution of the ordering protocol – a mechanism that is typically referred to as *pipelining*. However, this comes at the cost of using a software-based trusted computing base, rather than a hardware-based one akin the USIG service of MinBFT/MinZyzzyva.

2.3.5 Randomized Protocols

Randomized protocols are able to circumvent the FLP result discussed in Section 2.2 and solve agreement under the asynchronous system model using *randomization* (Ben-Or, 1983; Rabin, 1983). This approach also has the additional benefit of rendering the protocols completely leader-free and devoid of complicated corner-cases. However, the liveness guarantees

provided by these abstractions need to be weakened to allow a *probabilistic* termination of the protocol rather than a *deterministic* one. In the particular case of the consensus problem, the standard termination property needs to be re-written as "Every correct process eventually decides exactly one value *with probability 1*". By consequence, any total order broadcast protocol derived from this class of algorithm also provides probabilistic termination. In addition, these algorithms also require more communication steps and higher message complexity than their deterministic counterparts.

Randomized protocols are comprised by a non-deterministic mechanism that returns either 0 or 1, with equal probability. This mechanism is typically referred to as a *coin toss*. This coin-tossing mechanism can either be local to each process (Ben-Or, 1983), or distributed across processes, thus returning the same value at all correct ones (Rabin, 1983). Typically, local coin algorithms are simpler than their shared coin counterparts, but are expected to finish in an exponential number of rounds (Bracha, 1984). On the other hand, shared coin algorithms are expected (but not guaranteed) to finish in a constant number of rounds, but require sophisticated cryptographic schemes to safely implement coin sharing (Cachin *et al.*, 2005; Canetti & Rabin, 1993). Two of the most important works developed in this research area are SINTRA (Cachin & Poritz, 2002) and RITAS (Moniz *et al.*, 2011). Both works specify and implement a protocol stack that provides a total order broadcast primitive that can be used for SMR. More precisely, they guarantee total order by using a binary-value consensus algorithm and use a reliable broadcast protocol as the foundation for the stack. Those protocols are used to obtain a multi-value consensus algorithm, followed by second transformation from that multi-value algorithm to total order broadcast. The key difference between SINTRA and RITAS is the fact that SINTRA uses the shared coin algorithm by Cachin *et al.* (2005), whereas RITAS uses Bracha (1984) local coin algorithm.

2.3.6 Other Approaches

Aublin *et al.* (2015) proposed a well-defined modular abstraction unifying the optimizations proposed by previous protocols through composition, making it easy to design new protocols that are optimal in well-behaved executions (e.g., synchrony, absence of contention, no faults), but revert on-the-fly to PBFT if such behavior does not hold. Such modularity is at state machine replication level, in the sense that each module provides a way to totally order client requests under the aforementioned conditions.

Porto *et al.* (2015) introduced a new system model to design distributed algorithms dubbed Visigoth Fault Tolerance (VFT). In the VFT model, system assumptions are extended to account, not just for a limit of f total faults, but also for a limit of s slow but correct processes and o processes that may suffer from correlated commission faults. The authors show that when taking such additional assumptions into consideration, the total number of replicas required can be decomposed from $n \geq 3f + 1$ to $n \geq f + min(f, s) + o + 1$. Due to the use of these additional parameters, the VFT model is more flexible than the traditional asynchronous model, and as long as $f > min(f, s)$ and $f > o$, systems require less replicas to correctly secure liveness and safety properties, which in turn leads to better system performance. On the other hand, this model is less resilient against networks partition and assumptions violations: if the network is split in two parts, each one may continue executing concurrently and independently of each other. This is a situation that cannot happen in any of the three traditional models presented before. In addition, a violation of the limit s can lead to safety violations, whereas a violation of f in other models leads to only a loss of liveness.

2.4 Wide-Area Replication

The previous section reviewed the state of the art for SMR algorithms that, even though were designed for partially synchronous systems, are implemented and tested within local-area networks, where latency is low and predictable; they are not optimized for wide-area networks, were latency is high and variable. This section presents additional research that proposes SMR protocols targeting wide area deployments. We start by presenting a collection of wide-area protocols derived from Paxos, and then we proceed to discuss protocols designed to withstand Byzantine faults.

2.4.1 Protocols Derived From Paxos

One of the earliest works proposing an SMR protocol designed for wide-area was Mencius (Mao *et al.*, 2008), which forces replicas to take turn as the leader and propose client requests in their turns. Clients send requests to the replicas in their sites, which are submitted for ordering when a replica becomes the leader. According to the paper, this mechanism significantly reduces clients' latency in a WAN setting.

HP Paxos (Dobre *et al.*, 2010) is another wide-area SMR protocol that combines features of both classic and Fast Paxos (Lamport, 2006). Fast Paxos is able to finish each iteration in just two communication steps at the cost of requiring $3f + 1$ replicas instead of $2f + 1$. However, this is only possible in the absence of contention among clients. Otherwise, two additional communication step are necessary to finish an iteration. HP Paxos circumvents these limitations by employing operation history mechanisms inspired from Generalized Paxos (Lamport, 2005). The resulting protocol is still capable of finishing an iteration in two communication steps in the absence of collisions, while only requiring optimal number of replicas ($2f + 1$) and demanding the same number of communication steps of classic Paxos in the advent of collisions.

Kraska *et al.* (2013) proposed Multi-Data Center Consistency (MDCC), an optimistic transactional protocol that can execute across multiple, geographically distributed datacenters. Although its protocol is derived from Generalized Paxos (Lamport, 2005), its optimistic execution renders it able to execute without using a master replica, thus being able to terminate within a single communication round-trip when transactions do not conflict.

Egalitarian Paxos (EPaxos) (Moraru *et al.*, 2013) does not rely on a single designated leader for ordering operations. Instead, it enables clients to choose which replica should propose their operations, and employs a mechanism for solving conflicts between interfering operations. However, in order to correctly enforce such conflict resolution - and contrary to most SMR protocols - EPaxos requires information from the application to determine operations interference.

2.4.2 Protocols for the Byzantine Fault Model

Steward (Amir *et al.*, 2010) is a hierarchical Byzantine fault-tolerant protocol for multi-site systems. It is a hybrid algorithm in the sense that it runs a BFT agreement protocol within each site, and a lightweight, crash fault-tolerant protocol across sites. Since Steward assumes Byzantine failures, each site needs at least $3f_i+1$ replicas to run the BFT agreement protocol. But since it uses a lightweight protocol among sites, it is able to perform well in WANs. This comes at the cost of a complex protocol (over ten specialized algorithms that run within and among sites) that demands plenty of resources (each site requires $3f + 1$ replicas).

Mao *et al.* (2009) have addressed some challenges that rise when devising a BFT state machine protocol over WANs. They propose a system model comprised of multiple sites

similar to Steward's model, in which there is a lack of trust between sites; but unlike Steward, there is some trust within each site. Given this stronger assumption, they designed RAM, a protocol that takes advantage of 3 major concepts: (1) Mutually Suspicious Domains, which states that clients are able to trust servers within their own sites, whereas inter-site interactions are not trustworthy; (2) a rotating leader scheme similar to Mencius; (3) *Attested append-only memory* (A2M), a trusted system facility first introduced by Chun *et al.* (2007). This facility requires outgoing messages to be recorded into a trusted log, thus preventing hosts from lying in different ways to other hosts.

Following their previous work on MinBFT and Spinning, Veronese *et al.* (2010) introduced EBAWA, a BFT state machine replication protocol optimized for wide area networks. Since it uses the same USIG service that is used by MinBFT and MinZyzivva, it requires only $2f+1$ sites to tolerate f Byzantine faults. Each site can contain only one service replica, thus requiring the same communication pattern of MinBFT (illustrated in Figure 2.2b). Also like Spinning, it uses a rotating leader scheme to prevent a faulty leader from degrading system performance. This protocol presents the following advantages over both Steward and RAM: (1) it does not demand Steward's minimum of $3f_i + 1$ replicas in each site i and can survive compromised site; (2) whereas RAM requires the standard $3f + 1$ minimum number of sites, EBAWA requires only $2f + 1$ sites; and (3) since it supports the USIG trusted component, it does not require a log that may grow considerably and additional mechanism to append, look-up ans truncate entries like the A2M system needs.

A recent work by Miller *et al.* (2016) proposes Honeybadger, a leaderless BFT protocol designed for high-performance in cryptocurrency-oriented system. The key difference from Honeybadger to the above protocols discussed in this section is fact that it assumes a completely asynchronous system model, by adopting the same system model and randomization techniques discussed in Section 2.3.5. In particular, HoneyBadger is a protocol inspired by SINTRA (Cachin & Poritz, 2002) that is able to provide lower message complexity by means of an alternative algorithmic reduction based on the work of Ben-Or *et al.* (1994). In addition, HoneyBadger also adopts many optimizations that improve the performance of the system, such as a communication-optimal reliable roadcast (Cachin & Tessaro, 2005), the binary consensus proposed by Mostefaoui *et al.* (2015), and batching (Santos & Schiper, 2013b).

Finally, all the aforementioned algorithms explicitly separate the fault model from the network model. This approach results in protocols that are able to guarantee eventual progress,

as long as less than a third of the replicas are faulty – regardless of how many replicas are partitioned by an attacker.[1] By contrast, Liu *et al.* (2016) propose a novel SMR approach dubbed XFT, which introduces a limit on the amount of partitioned replicas present in the system. This limit enabled the creation of XPaxos, an SMR protocol intended for wide-area replication that requires $2f+1$ replicas under the assumption that a majority of these replicas remains both correct and synchronous.

2.5 Concluding Remarks

In this section we discussed the context for this thesis and presented the background for BFT state machine replication. In particular, we have described why the consensus problem is heavily related to this technique. We also presented important milestones such as the FLP impossibility and ways to circumvent it, the PBFT protocol and the emergence of the research area, as well as many important ideas such as hybrid protocols and randomization. We also provided a brief overview of the work dedicated to deploy the SMR technique into geo-replicated settings. For the remaining of the thesis, we will focus on describing the work developed and the results obtained.

[1] In this context, a partitioned replica is one that is unable to reach other replicas within the time bounds define for the network.

3

Mod-SMaRt

This chapter presents the thesis first step towards a robust SMR codebase in the form of Mod-SMaRt (Modular State Machine Replication), a modular algorithm built directly around an augmented Byzantine consensus primitive dubbed VP-Consensus (Validated and Provable Consensus). Mod-SMaRt requires a minimal number of communication steps between replicas, thus being an optimal transformation from Byzantine consensus to BFT-SMR.

The chapter is organized in the following way. We first describe the problem at hand in Sections 3.1 and 3.2. In Section 3.3 we unpack the system model adopted for both VP-Consensus and Mod-SMaRt. The VP-Consensus primitive is discussed in Section 3.4 and the Mod-SMaRt algorithms is described in Section 3.5. Possible optimizations and additional considerations are discussed in Section 3.6. In Sections 3.7 and 3.8 we discuss additional related work and present our concluding remarks, respectively.

3.1 Modular vs Monolithic algorithms

In the last decade, many practical SMR protocols for the Byzantine fault model were published (e.g., Abd-El-Malek *et al.* (2005); Castro & Liskov (2002); Cowling *et al.* (2006); Kotla *et al.* (2009); Veronese *et al.* (2009)). However, despite their efficiency, such protocols are *monolithic*: they do not clearly separate the consensus primitive from the remaining of the protocol. Moreover, The implementation of these algorithms adds even more complexity to these protocols – including for the crash fault model, as was observed in (Chandra *et al.*,

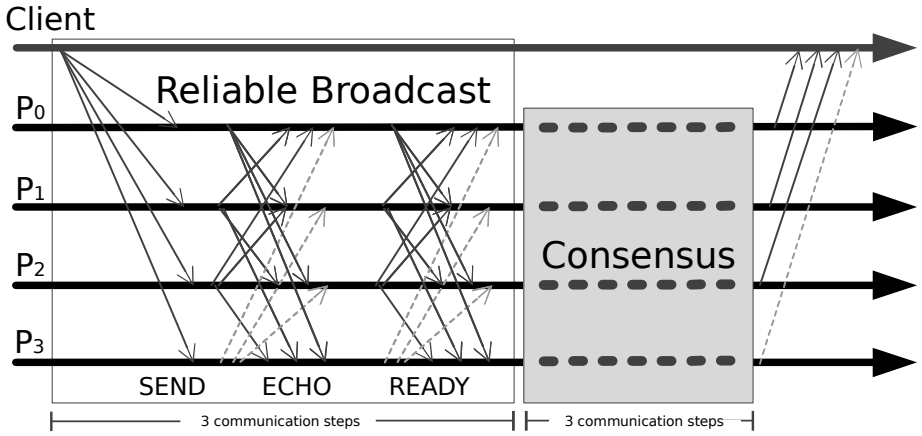


Figure 3.1: Modular BFT SMR message pattern for a protocol that uses reliable broadcast and a consensus primitive. This protocol is adapted from (Milosevic *et al.*, 2011).

2007). Besides being hard to evaluate such implementations, its even harder to verify their correctness.

By contrast, many of the BFT total order broadcast protocols proposed in the literature (the main component of a BFT SMR implementation) are obtained by means of a protocol stack, i.e., they are built using black-box Byzantine consensus primitives (e.g., Cachin *et al.* (2001); Correia *et al.* (2006); Hadzilacos & Toueg (1993); Milosevic *et al.* (2011)). This modularity simplifies the protocols, making them easier to both understand and implement. However, these modular transformations plus the underlying consensus they use always require more communication steps than the aforementioned monolithic solutions.

To better understand the difference between them, Figure 3.1 presents the typical message pattern of modular BFT total order broadcast protocols when used to implement SMR. The key point of most of these transformations is the use of BFT reliable broadcast protocol (Bracha, 1984) to disseminate clients' requests among replicas, ensuring they will be eventually proposed (and decided) in some consensus instance that defines the order of messages to be executed. As illustrated in Figure 3.1, the usual BFT reliable broadcast requires 3 communication steps (Bracha, 1984).[1]

[1] Alternatively, some known transformations require the use of an echo broadcast (Toueg, 1984) instead of a reliable broadcast, thus requiring two extra communication steps instead of three. However, those transformations require the replicas to sign each exchanged messages (Cachin *et al.*, 2001; Doudou *et al.*, 2005), being thus even more inefficient, specially in local area networks.

Finally, it is known that optimally resilient Byzantine consensus protocols cannot safely decide a value in 2 or less communication steps (Dutta *et al.*, 2005; Martin & Alvisi, 2006). This means that latency-optimal protocols for BFT SMR that use only $3f + 1$ replicas to tolerate f Byzantine faults needs to execute at least 3 communication steps for the consensus, plus 3 more steps to receive the request from the client and send a reply.[1] The consequence is that the protocol of Figure 3.1 requires at least 6 communication steps to totally order a message in the best-case, plus one more to send a reply to the client, making a total of 7 steps. By comparison, PBFT (Castro & Liskov, 2002) requires only 5 communication steps (3 for the embedded Byzantine consensus plus 2 for client-replica communication – see Figure 2.2a) in the best-case. Monolithic protocols, as already stated, circumvent those extra steps by mixing the reliable broadcast with the consensus primitive, at the cost of becoming considerably more complex than a modular approach.

3.2 Preserving Robustness, Modularity and Latency

The observation of previous section is strongly linked to the objectives of this thesis. More precisely, since our intention is to provide a efficient SMR protocol implemented in a robust codebase, the first step should be to devise a protocol that preserves optimal latency (i.e., the minimum number of communication steps) while still offering the modularity needed to avoid the complexity of monolithic protocols.

The solution proposed in this chapter is a transformation from Byzantine consensus to BFT SMR which uses an augmented Byzantine consensus primitive. This primitive is called *Validated and Provable Consensus* (VP-Consensus) and the resulting transformation is a BFT SMR algorithm dubbed *Modular State Machine Replication* (Mod-SMaRt). VP-Consensus is used as a *"grey-box" abstraction* that allows modular implementation of SMR without using reliable broadcast, thus avoiding the extra communication steps required to safely guarantee that all requests arrive at all correct replicas. Moreover, this primitive can be easily obtained by modifying existing leader-driven consensus algorithms (e.g., Cachin (2009); Lampson (2001); Martin & Alvisi (2006); Rütti *et al.* (2010); Zielinski (2004)). By clearly separating the Byzantine consensus from the rest of the protocol, we simplify its

[1] This excludes optimistic protocols that are very efficient in contention-free executions (Abd-El-Malek *et al.*, 2005; Cowling *et al.*, 2006), speculative protocols (Kotla *et al.*, 2009) and protocols requiring more than $3f + 1$ replicas (Martin & Alvisi, 2006).

specification (thus, making it easier to prove correct), and encapsulate most of the complexity within a consensus primitive (making it easier to understand by developers and users).

Mod-SMaRt avoids mixing protocols through the use of a well-defined interface exported by VP-Consensus, that allows it to handle request' timeouts and, if needed, triggers internal consensus timeouts as necessary. We see the use of an augmented consensus primitive as a good trade-off between modularity and efficiency, specially when this primitive can be easily supported with simple modifications on several leader-driven partially-synchronous Byzantine consensus protocols (Cachin, 2009; Lampson, 2001; Li *et al.*, 2007; Martin & Alvisi, 2006; Rütti *et al.*, 2010; Zielinski, 2004). Moreover, even perfect black-box modular transformations from Byzantine consensus to total order broadcast requires the consensus module to satisfy special *Validity* properties to ensure that the decided value was proposed by a correct process — or at least to ensure it is valid with respect to the remaining of the algorithm (Milosevic *et al.*, 2011).

3.3 System Model

We consider a system composed by a set of $n \geq 3f + 1$ replicas R, where a maximum of f replicas may be subject to Byzantine faults, and a set C with an unbounded (but finite) number of clients, which can also suffer Byzantine faults.

Like in PBFT and similar protocols (Castro & Liskov, 2002; Cowling *et al.*, 2006; Kotla *et al.*, 2009; Veronese *et al.*, 2009), Mod-SMaRt does not require synchrony to assure *safety*. However, it requires synchrony to provide *liveness*. This means that, even in the presence of faults, correct replicas will never evolve into an inconsistent state; but the execution of the protocol shall terminate only when the system becomes synchronous. Due to this, we assume an *eventually synchronous* system model (Dwork *et al.*, 1988). As discussed in Section 2.1.1, this model assumes the system operates asynchronously until some unknown instant, at which it will become synchronous. At this point, time bounds for computation and communication (which are also unknown) shall be obeyed by the system.

We further assume the existence of authenticated perfect point-to-point links as described in Section 2.1.1, cryptographic functions that provide digital signatures, message digests, and message authentication codes (MAC).

3.4 Validated and Provable Consensus

In this section we introduce the concept of *Validated and Provable Consensus* (VP-Consensus). By 'Validated', we mean the protocol receives a predicate γ together with the proposed value — which any decided value must satisfy. By 'Provable', we mean that the protocol generates a cryptographic proof Γ that certifies that a value v was decided in a consensus instance i. More precisely, a VP-Consensus implementation offers the following interface:

- *VP-Propose*(i, l, γ, v): proposes a value v in consensus instance i, with initial leader l and predicate γ;
- *VP-Decide*(i, v, Γ): triggered when value v with proof Γ is decided in consensus instance i;
- *VP-Timeout*(i, l): Used to trigger a timeout in the consensus instance i, and appoint a new leader process l. More details about the use of this procedure will be presented in Section 3.5.5.

Three important things should be noted about this interface. First, VP-Consensus assumes a leader-driven protocol, in the same sense as any Byzantine Paxos consensus primitive. Second, the interface assumes the VP-Consensus implementation can handle timeouts to change leaders, and a new leader is (locally) chosen after a timeout. The need for these two requirements will be explained latter. Finally, we implicitly assume that all correct processes will invoke *VP-Propose* for an instance i using the same predicate γ.

Besides respecting the classic properties described in Section 2.2 (Termination, Agreement, and Integrity), VP-Consensus also introduces the following additional properties:

- *External Validity*: If a correct process decides v, then $\gamma(v)$ is true;
- *External Provability*: If some correct process decides v with proof Γ in consensus instance i, any correct process can verify that v is the decision of i through Γ.

External Validity was originally proposed by Cachin et al. (Cachin *et al.*, 2001), but we use a slightly modified definition. In particular, *External Validity* no longer explicitly demands validation data for proposing v, because such data is already included in the proposed value, as will be clear in Section 3.5.

3.4.1 Implementation requirements

Even though our primitive offers the classical properties of consensus, the interface imposes some changes in its implementation. Notice that we are not trying to specify a new consensus algorithm; we are only specifying a primitive that can be obtained by making simple modifications to existing ones. However, as described before, our interface assumes that such algorithms are leader-driven and meant for the partially synchronous system. Most Paxos-based protocols satisfy these conditions (Cachin, 2009; Lamport, 2006; Martin & Alvisi, 2006; Rütti *et al.*, 2010; Zielinski, 2004), and thus can be used with Mod-SMaRt. In this section we present an overview of the required modifications on consensus protocols, without providing explanations for it. We will come back to the modifications in Section 3.5.5, when it will become clear why they are required.

The first change is related to the timers needed in the presence of partial synchrony. To our knowledge, all algorithms in literature for such system model requires a timer associated to such mechanism (Cachin, 2009; Lamport, 1998, 2006; Martin & Alvisi, 2006). The primitive still needs such timer; but it will not be its responsibility to manage it (or even stop it). Instead, we use the procedure *VP-Timeout* to indicate to the consensus that a timeout has occurred, and it needs to be handled.

The second change is related to the assumption of a leader-driven consensus. To the best of our knowledge, all the leader-driven algorithms in the literature have deterministic mechanisms to select a new leader when sufficiently many of them suspect the current one. These suspicions are triggered by a timeout. A VP-Consensus implementation still requires the election of a new leader upon a timeout. However, the next leader will be defined by Mod-SMaRt, and is passed as an argument in the *VP-Propose* and *VP-Timeout* calls. Notice that these first two requirements are equivalent to assuming the consensus protocol requires a leader election module, just like Ω failure detector, which is already required in some algorithms (Cachin, 2009; Martin & Alvisi, 2006).

The third change imposes the consensus algorithm to generate the cryptographic proof Γ to fulfill the *External Provability* property. This proof can be generated by signing the messages that can trigger a decision of the consensus.[1] An example of proofs would be a set

[1] Due to the efficiency cost of producing digital signatures, the cryptographic proof can be generated with MAC vectors instead of digital signatures, just like in PBFT (Castro & Liskov, 2002).

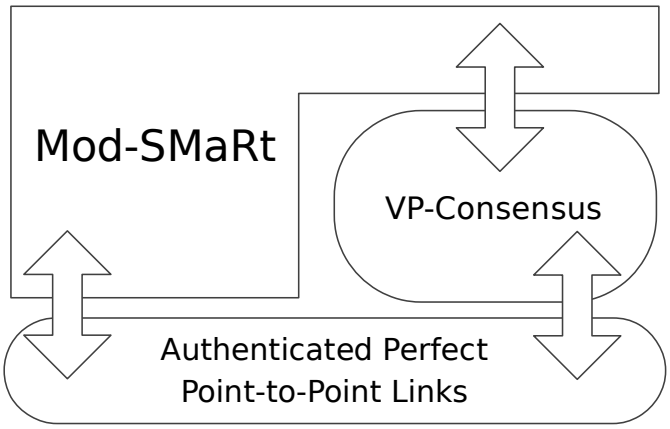


Figure 3.2: Mod-SMaRt replica architecture. The authenticated perfect point-to-point links guarantee the delivery of replica-to-replica messages, while the VP-Consensus module is used to establish agreement on the message(s) to be delivered by consensus instances.

of $2f + 1$ signed COMMIT messages in PBFT (Castro & Liskov, 2002) or $\lceil \frac{n+f+1}{2} \rceil$ signed COMMITPROOF messages in Parametrized FaB (Martin & Alvisi, 2006).

Finally, we require each correct process running the consensus algorithm to verify if the value being proposed by the leader satisfies γ before it is accepted. Correct processes must only accept values that satisfy such predicate and discard others — thus fulfilling the *External Validity* property.

3.5 The Mod-SMaRt Algorithm

In this section we describe Mod-SMaRt, our modular BFT SMR algorithm, which is divided into three sub-algorithms: client operation, normal phase, and synchronization phase. The full proofs showing that Mod-SMaRt satisfies the properties of a BFT SMR under our system model are presented in Appendix A.

3.5.1 Overview

The general architecture of a replica is described in Figure 3.2. Mod-SMaRt is built on top of an authenticated perfect point-to-point links communication layer, as well as the VP-Consensus module described in previous section. Such module may also use the same

communication support to exchange messages among processes. Mod-SMaRt uses VP-Consensus to execute a sequence of consensus instances, where in each instance i a batch of operations are proposed for execution, and the same proposed batch is decided on each correct replica. This is the mechanism by which we are able to achieve total order across correct replicas.

During normal phase, a log of the decided values is constructed by the sequence of VP-Consensus executions. Each log entry contains the decided value, the *id* of the consensus instance where it was decided, and its associated proof. To simplify our design, Mod-SMaRt assumes that each correct replica can execute concurrently only the current instance i and previous consensus instance $i-1$ at any given moment. All correct replicas remain available to participate in consensus instance $i-1$, even if they are already executing i. This is done to ensure that if there is one correct replica running consensus $i-1$ but not i, there will be at least $2f+1$ correct replicas executing $i-1$, which ensures the delayed replica will be able to finish $i-1$.

Due to the expected asynchrony of the system, it is possible that a replica receives messages for a consensus instance j such that $j > i$ (early message) or $j < i-1$ (outdated message). Early messages are stored in an out-of-context buffer for future processing while outdated messages are discarded. Notice that we do not provide pseudo-code for this mechanism, relying on our communication layer to deliver messages in accordance with the consensus instances being executed.

This pretty much describes the *normal phase* of the protocol, which is executed in the absence of faults and in the presence of synchrony. When these conditions are not satisfied, the *synchronization phase* might be triggered. Moreover, Mod-SMaRt makes use of the concept of *regencies*. This is equivalent to the *view* mechanism employed by PBFT and ViewStamped Replication (Castro & Liskov, 2002; Oki & Liskov, 1988), where a single replica will be assigned as the leader for each regency. Such leader will be needed both in Mod-SMaRt, and in the VP-Consensus module. During each regency, the normal case operation can be repeated infinitely; during a synchronization phase, an unbounded (but finite) number of regency changes can take place, since the system will eventually become synchronous.

Since we avoid using reliable multicast before running the Byzantine consensus protocol, two problems may happen. First, a malicious leader can make a client starve by not proposing messages from that client for ordering. Second, a malicious client can send messages

to all replicas but to the current (correct) leader, making other processes suspect it for not proposing messages from this client. The solution for these problems is to suspect the leader only if the timer associated with a message expires twice, making the process forward the pending message to the leader upon the first expiration.

In case a regency change is needed, timeouts will be triggered at all correct replicas and the synchronization phase will take place. During this phase, Mod-SMaRt must ensure three things: (1) a quorum of $n - f$ replicas must have the pending messages that caused the timeouts; (2) correct replicas must exchange logs to reach the same consensus instance; and (3) a timeout is triggered in this instance, proposing the same leader at all correct replicas (the one chosen during the regency change). Notice that Mod-SMaRt does not verify consensus values to ensure consistency: all these checks are done inside of the VP-Consensus module, after its timeout is triggered. This substantially simplifies faulty leader recovery by breaking the problem in two self-contained blocks: SMR layer ensures all processes are executing the same consensus with the same leader while VP-Consensus deals with the inconsistencies caused by a faulty leader.

3.5.2 Client Operation

Algorithm 1 describes how the client invokes an operation in Mod-SMaRt. This algorithm can be seen as a synchronous call of the $Invoke$ function that will broadcast a given request to all the replicas in the system and wait for replies from a quorum.

When a client wants to issue a request to the replicas, it sends a $\mathrm{REQUEST}$ message in the format specified in line 9. This message contains the sequence number for the request, the operation issued by the client, and a boolean parameter that indicates if the message contains a read-only operation (i.e., an operation that does not modify the application's state). The inclusion of a sequence number uniquely identifies the operation (together with the client id), and prevent replay attacks made by an adversary that might be sniffing the communication layer. A digital signature α_c is appended to the message to prove that such message was produced by client c. Although this signature could be eschewed in favor of a MAC vector, its use makes the system resilient against certain attacks (Amir *et al.*, 2011).

The client waits for at least $\lceil \frac{n+f+1}{2} \rceil$ matching replies from different replicas, for the same sequence number, as it can be observed in lines 12–13, when the result is returned in

Algorithm 1: Algorithm at client c

1 **Upon** *Init* **do**
2 $nextSeq \leftarrow 0$
3 $requests[1..\infty]$
4 $\forall i \in \mathbb{N}_0 : requests[i] \leftarrow \emptyset$
5 $Replies \leftarrow \emptyset$

6 **Upon** *Invoke(op, readOnly)* **do**
7 $nextSeq \leftarrow nextSeq + 1$
8 $requests[nextSeq] \leftarrow \langle op, readOnly \rangle$
9 `send` $\langle \text{REQUEST}, nextSeq, op, readOnly \rangle_{\alpha_c}$ to R

10 **Upon** reception of $\langle \text{REPLY}, seq, rep \rangle$ from $r \in R$ **do**
11 $Replies \leftarrow Replies \cup \{\langle r, seq, rep \rangle\}$
12 **if** $\exists\, seq, rep : |\{\langle *, seq, rep \rangle \in Replies\}| >= \lceil \frac{n+f+1}{2} \rceil$
13 $Replies \leftarrow Replies \backslash \{\langle *, seq, rep \rangle\}$
14 **return** rep
15 **if** $\exists\, seq, op : (|\{\langle *, seq, * \rangle \in Replies\}| >= \lceil \frac{n+f+1}{2} \rceil \wedge requests[seq] = \langle op, TRUE \rangle)$
16 $Invoke(op, FALSE)$

line 14. In case the client invoke a request for a read-only operation but there are no $\lceil \frac{n+f+1}{2} \rceil$ matching replies, the request is re-transmitted as a standard request (lines 15–16).

3.5.3 Normal Phase

The goal of the normal phase is to execute a sequence of consensus instances in each replica. The values proposed by each replica will be a batch of operations issued by the clients. Because each correct replica executes the same sequence of consensus instances, the values decided in each instance will be the same in all correct replicas, and since they are batches of operations, they will be *totally ordered* across correct replicas. All variables and functions used in Algorithms 2-4 are described in Table 3.1.

Figure 3.3 illustrates this phase message pattern and Algorithm 2 formally describes it. Reception of client requests are processed in line 1-2 through procedure $RequestReceived$ (lines 21–25). Requests are only considered by correct replicas if the message contains a valid signature, a valid operation, and the sequence number expected from this client (to

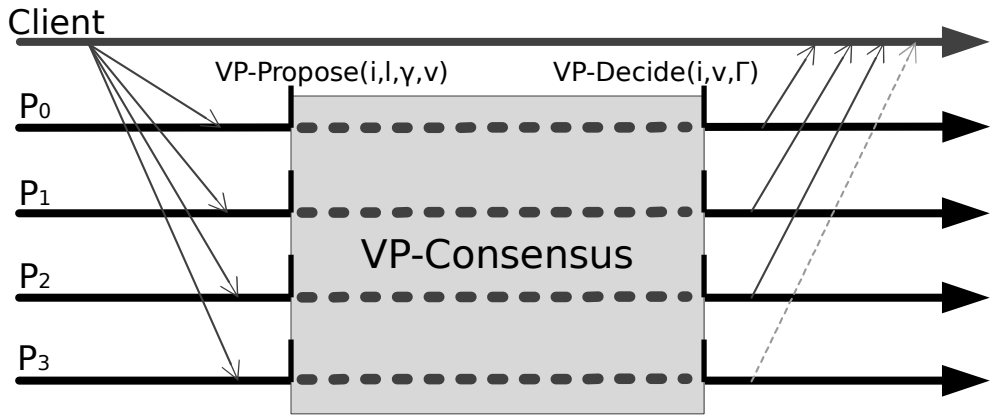


Figure 3.3: Communication pattern of Mod-SMaRt' normal phase for $f = 1$. Each client sends its operations to the replicas, a consensus instance is immediately started, and the decided value is sent to the client.

avoid replay attacks). If a replica accepts an operation issued by a client, it stores it in the $ToOrder$ set, activating a timer associated with the request (lines 23–25). Notice that a message is also accepted if it is forwarded by other replicas (lines 18–20).

When the $ToOrder$ set contains some request to be ordered, there is no consensus being executed and the ordering of messages is not stopped (see next section), a sub-set of operations $Batch$ from $ToOrder$ is selected to be ordered (lines 3–4). The predicate *fair* ensures that all clients with pending requests will have approximately the same number of operations in a batch to avoid starvation. The replica will then create a consensus instance, using $Batch$ as the proposed value (lines 5–6). The predicate γ given as an argument in the *VP-Propose* procedure should return $TRUE$ for a proposed value V if the following three conditions are met: (1) $fair(V)$ is $TRUE$ (thus V is not an empty set); (2) Each message in V is either in the $ToOrder$ set of the replica or is correctly signed and contains the next sequence number expected from the client that issued the operation; and (3) $validCmd(m)$ is $TRUE$ for all messages contained in V.

Algorithm 2: Normal phase at replica r.

Upon reception of $m = \langle \text{REQUEST}, seq, op, FALSE \rangle \alpha_c$ from $c \in C$ **do**
  `RequestReceived` (c, m)

Upon $(toOrder \neq \emptyset) \wedge (currentCons = -1) \wedge (\neg stopped)$ **do**
  $Batch \leftarrow X \subseteq ToOrder : |X| \leq maxBatch \wedge$ `fair`(X)
  $currentCons \leftarrow$ `highCons`$(DecLog).i + 1$
  `VP-Propose` $(currentCons, creg \bmod R, \gamma, Batch)$

Upon *VP-Decide*$\langle i, Batch, Proof \rangle$ **do**
  if $stopped$
    $Decided \leftarrow Decided \cup \{\langle i, Batch, Proof \rangle\}$
  else
    $DecLog \leftarrow DecLog \cup \{\langle i, Batch, Proof \rangle\}$
    if $currentCons = i$ **then** $currentCons \leftarrow -1$
    `// Deterministic cycle`
    foreach $m = \langle \text{REQUEST}, seq, op, readOnly \rangle_{\alpha_c} \in Batch$ **do**
      `cancelTimers` $(\{m\})$
      $ToOrder \leftarrow ToOrder \backslash \{m\}$
      $rep \leftarrow$ `execute`(op)
      `send` $\langle \text{REPLY}, seq, rep \rangle$ to c

Upon reception of $\langle \text{FORWARDED}, M \rangle$ from $r' \in R$ **do**
  foreach $m = \langle \text{REQUEST}, *, *, FALSE \rangle \alpha_c \in M$ **do**
    `RequestReceived` (c, m)

Procedure *RequestReceived*(c, m)
  if $lastSeq[c] + 1 = m.seq \wedge$ `validSig`$(m) \wedge$ `validCmd`(m)
    $ToOrder \leftarrow ToOrder \cup \{m\}$
    if $\neg stopped$ **then** `activateTimers`$(\{m\}, timeout)$
    $lastSeq[c] \leftarrow m.seq$

Upon reception of $m = \langle \text{REQUEST}, seq, op, TRUE \rangle \alpha_c$ from $c \in C$ **do**
  $rep \leftarrow$ `execute`(op)
  `send` $\langle \text{REPLY}, seq, rep \rangle$ to c

Variables

Name	Initial Value	Description
$timeout$	INITIAL_TIMEOUT	Initial timeout value
$maxBatch$	MAX_BATCH	Maximum number of operations that a batch may contain
$creg$	0	Replica's current regency
$nreg$	0	Replica's next regency
$currentCons$	-1	Current consensus being executed
$DecLog$	$\emptyset$	Replica's log of all consensus instances and their proofs
$ToOrder$	$\emptyset$	Operations to be ordered
$Stops$	$\emptyset$	Requests collected in STOP messages
$Decided$	$\emptyset$	Decision value obtained during the synchronization phase
$stopped$	FALSE	Indicates if the synchronization phase is activated or not
$lastSeq[1..\infty]$	$\forall c \in C : lastSeq[c] \leftarrow 0$	Last sequence number sent by each client
$ChangeReg[1..\infty]$	$\forall g \in \mathbb{N} : ChangeReg[g] \leftarrow \emptyset$	Replicas that want a regency change
$Data[1..\infty]$	$\forall g \in \mathbb{N} : Data[g] \leftarrow \emptyset$	STOPDATA messages collected by the leader
$Sync[1..\infty]$	$\forall g \in \mathbb{N} : Sync[g] \leftarrow \emptyset$	Set of logs sent by the leader to all replicas

Functions

Interface	Description
`activateTimers(Reqs, timeout)`	Creates a timer for each request in *Reqs* with value *timeout*
`cancelTimers(Reqs)`	Cancels the timer associated with each request in *Reqs*
`execute(op)`	Makes the application execute operation *op*, returning the result
`noGaps(Log)`	Returns TRUE if sequence of consensus *Log* does not contain any gaps
`highCons(Log)`	Returns the consensus instance from *Log* with highest id
`highLog(Logs)`	Returns the most advanced sequence of consensus contained in *Logs*
`validDec(decision)`	Returns TRUE if *decision* contains a valid proof
`validSig(req)`	Returns TRUE if request *req* is correctly signed
`validCmd(req)`	Returns TRUE if request *req* contains a valid operation with respect to the application The implementation of this function is provided by the developers

Table 3.1: Variables and functions used in Algorithms 2, 3, and 4.

When a consensus instance decides a value (i.e., a batch of operations) and produces its corresponding proof (line 7), Mod-SMaRt will: store the batch of operations and its cryptographic proof in each replica's log (line 11); cancel the timers associated with each decided request (line 14); deterministically deliver each operation contained in the batch to the application (line 16); and send a reply to the client that requested the operation with the corresponding result (line 17). Notice that if the algorithm is stopped (because the replica is running a synchronization phase, see next section), decided messages are stored in a $Decided$ set (lines 8 and 9), instead of being executed.

Finally, if a replica receives a read-only operation, it immediately executes it and sends the reply to the client (lines 26–28). Since read-only operations do not modify the application state, they are not required to be totally ordered and can be executed instantly.[1]

[1]It is the responsibility of the developer to identify which operations can be classified as read-only. If operations are miss-classified, Mod-SMaRt cannot guarantee consistency across replicas.

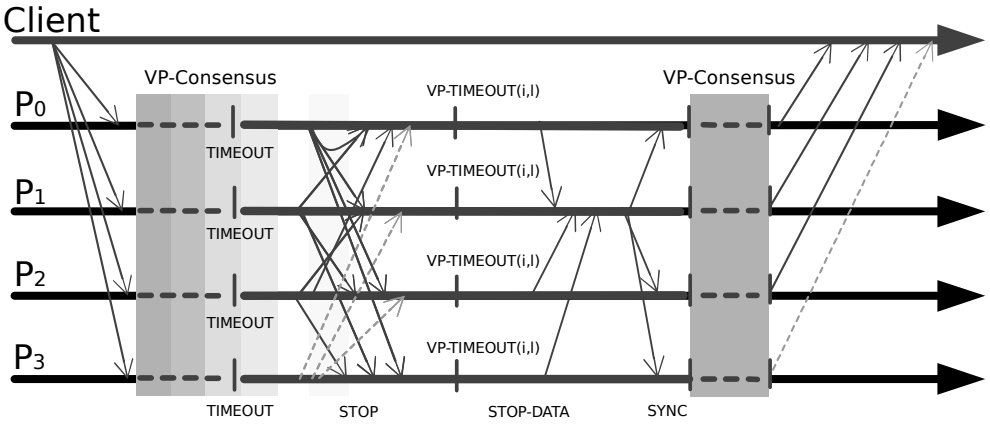


Figure 3.4: Communication pattern of synchronization phase for $f = 1$. This phase is started when the timeout for a message is triggered for a second time.

3.5.4 Synchronization Phase

The synchronization phase is described in Algorithms 3-4, and its message pattern is illustrated in Figure 3.4. This phase performs a regency change and force correct replicas to synchronize their states and go to the same consensus instance. It occurs when the system is passing through a period of asynchrony, or there is a faulty leader that does not deliver client' requests before their associated timers expire. This phase is started when a *timeout* event is triggered for a sub-set M of operations in $ToOrder$ that were not ordered (line 1).

When the timers associated with a set of requests M are triggered for the first time, the requests are forwarded to all replicas (lines 2–3). This is done because a faulty client may have sent its operation only to some of the replicas, therefore starting a consensus in less than $2f + 1$ of them. This step forces such requests to reach all correct replicas if clients behave this way, without forcing a leader change. If there is a second timeout for the requests, the replica starts a regency change (line 4). When a regency change begins in a replica, the timers for all pending requests are cancelled (line 8) and a STOP message is sent to all replicas (line 9). This message informs other replicas that a timeout for a given set of requests has occurred. When a replica receives more than f STOP messages ordering the next regency to be installed (line 14), it begins to change its current regency using the valid messages in $Stops$ (line 15), even if no timeout is triggered locally. This mechanism ensures that at least one correct replica has begun a regency change, and therefore this replica can safely begin to change its regency.

Algorithm 3: Synchronization phase at replica r (part 1).

```
1  Upon timeout for requests M do
2      M_first ← {m ∈ M : first timeout of m}
3      if M_first ≠ ∅ then send ⟨FORWARDED, M_first⟩ to R
4      if M\M_first ≠ ∅ then StartRegChange (M\M_first)

5  Procedure StartRegChange(M)
6      if nreg = creg
7          nreg ← creg + 1
8          cancelTimers(ToOrder) // Cancel all timers
9          send ⟨STOP, nreg, M⟩ to R

10 Upon reception of ⟨STOP, reg, M⟩ from r' ∈ R do
11     if reg = creg + 1
12         Stops ← Stops ∪ M
13         ChangeReg[reg] ← ChangeReg[reg] ∪ {r'}
14         if |ChangeReg[reg]| > f
15             M' ← {m ∈ Stops : m.seq > lastSeq[m.c] ∧ validSig(m)}
16             StartRegChange (M')
17             ToOrder ← ToOrder ∪ M'
18             if |ChangeReg[reg]| > 2f ∧ reg = nreg
19                 stopped ← TRUE
20                 creg ← nreg
21                 leader ← creg mod |R|
22                 if currentCons ≠ −1
23                     VP-Timeout (currentCons, leader)
24                 activateTimers (ToOrder, timeout)
25                 send ⟨STOPDATA, reg, DecLog⟩_{α_r} to leader
```

When a replica receives more than $2f$ STOP messages, the processing of decisions is stopped (line 19), the new regency is installed (line 20), a new leader is elected (line 21), and a timeout is triggered at the consensus instance being executed (lines 22–23) . It is necessary to wait at for least $2f+1$ messages to make sure that eventually all correct replicas install the next regency and that the VP-Consensus primitive correctly handles its timeouts. Following this, the timers for all operations in the $ToOrder$ set are re-activated and a new leader is elected (lines 23–24).

Algorithm 4: Synchronization phase at replica r (part 2).

```
26 Upon reception of m = ⟨STOPDATA, creg, Log⟩_{α_r} from r' ∈ R do
27     if creg mod n = r
28         if (noGaps(Log)) ∧ (∀d ∈ Log : validDec(d))
29             Data[creg] ← Data[creg] ∪ {m}
30         if |Data[creg]| ≥ n − f
31             send ⟨SYNC, creg, Data[creg]⟩ to R

32 Upon reception of ⟨SYNC, creg, Proofs⟩ from r' ∈ R do
33     if (creg mod n = r')
34         foreach ⟨STOPDATA, creg, Log⟩_{α_r''} ∈ Proofs do
35             if (noGaps(Log)) ∧ (∀d ∈ Log : validDec(d))
36                 Sync[creg] ← Sync[creg] ∪ {⟨r'', Log⟩}
37         if |Sync[creg]| ≥ n − f
38             L ← highLog(Sync[creg] ∪ {⟨r, (DecLog ∪ Decided)⟩})
39             if currentCons < highCons(L).i
40                 currentCons = −1
41             Stops ← ∅
42             Decided ← ∅
43             stopped ← FALSE
               // Deterministic cycle
44             foreach ⟨i', B, P⟩ ∈ L : i' > highCons(DecLog).i do
45                 Trigger VP-Decide⟨i', B, P⟩
```

After the next regency is installed, it is necessary to force all replicas to go to the same state (i.e., synchronize their logs and execute the logged requests) and, if necessary, start the consensus instance. To accomplish this, all replicas send a STOPDATA message to the new regency' leader, providing it with their decision log (line 25). As long as the proof associated with each decided value is valid and there is no consensus instance missing, the leader will collect these messages (lines 28–29). This is necessary because it proves that each consensus instances has decided some batch of operations (which will be important later). When at least $n - f$ valid STOPDATA messages are received by the leader, it will send a SYNC message to all replicas, containing all the information gathered about their decided instances in at least $n - f$ replicas (lines 30–31).

When a replica receives the SYNC message, it executes the same computations performed by the leader (lines 33–37). This is necessary to ensure that the leader has gathered and sent valid information. If the leader is correct, after receiving the same SYNC message, all correct replicas will choose the same highest log (line 38) and, in case the replica is not executing the lattest consensus instance, make everything ready to start at that instance (line 39-40). All correct replicas resume decision processing (line 43) and evolve into the same state, as they deliver the value of each consensus instance that was already decided in other replicas (lines 44–45).

3.5.5 Reasoning about the Consensus Modifications

As we mentioned in Section 3.4.1, the VP-Consensus primitive does not need to manage (i.e., start and stop) timers, since our SMR algorithm already implements them. Due to this, the VP-Consensus module only needs to be notified by the SMR algorithm when it needs to handle a timeout. This is done by invoking *VP-Timeout* for a consensus i, when installing a new regency (lines 22-23 of Algorithm 3). The *VP-Timeout* operation also gives as an argument the new leader that the replica should rely on. This is needed because we assume a leader-driven consensus, and such algorithms tend to elect the leader in a coordinated manner. But when a delayed replica jumps to a consensus i during the synchronization phase, it will be out-of-sync with respect to the current regency, when compared with the majority of replicas that have already started consensus i during the normal phase. For this reason, we need to explicitly inform VP-Consensus about the new leader.

Let us now discuss why the *External Validity* is required for Mod-SMaRt. The classic *Validity* property would be sufficient in the crash fault model, because processes are assumed to fail only by stopping, and will not propose invalid values; however, in the Byzantine fault model such behavior is possible. A faulty process may propose an invalid value, and such value might be decided. An example of such value can be an empty batch. This is a case that can prevent progress within the algorithm. By forcing the consensus primitive to decide a value that is useful for the algorithm to keep making progress, we can prevent such scenario from occurring, and guarantee liveness as long as the execution is synchronous.

Finally, it should now be clear why the *External Provability* property is necessary: in the Byzantine fault model, replicas can lie about which consensus instance they have actually finished executing, and also provide a fake/corrupted decision value if a synchronization

phase is triggered. By forcing the consensus primitive to provide a proof, we can prevent faulty replicas from lying. The worst thing a faulty replica can do, is to send old proofs from previous consensus, but because Mod-SMaRt requires at least $n - f$ logs from different replicas, there will be always more than f up-to-date, correct replicas that will provide their most recent consensus decision.

3.5.6 Mod-SMaRt for Crash Faults Only

Even though Mod-SMaRt is designed for Byzantine fault tolerance, the protocol can be easily adapted to crash fault tolerance, where the theoretical lower bound on the number of replicas is $2f + 1$ (Lamport, 1998; Oki & Liskov, 1988).

The first modification requires usage of simple majority quorums instead of Byzantine dissemination quorums (Malkhi & Reiter, 1998). This can be done by changing lines 12 and 15 in Algorithm 1 to wait for $\lceil \frac{n+1}{2} \rceil$ replies instead of $\lceil \frac{n+f+1}{2} \rceil$. Moreover, Algorithm 3 would required a single STOP message to agree to participate in the synchronization phase (Algorithm 3, line 14) and $f + 1$ STOP messages to install a new regency and proceed with the synchronization phase (lines 18–25). The above modifications can be done because, since replicas only fail by stopping to execute, each individual process is allowed to trust any message that it receives from its peers. For this same reason, Mod-SMaRt no longer requires digital signatures either for REQUEST or STOPDATA messages. Finally, the protocol no longer requires a consensus primitive to be either verifiable or provable.

3.6 Optimizations

In this section we discuss a set of optimizations for an effective implementation VP-Consensus and Mod-SMaRt and discuss how these optimizations can result in a protocol similar to the classic PBFT (Castro & Liskov, 2002).

3.6.1 Symmetric Cryptography

The first optimization aims to avoid the computational cost of generating and verifying digital signatures: client request can use MAC vectors instead of digital signatures, as is done in PBFT. However, this results in a less robust SMR implementation vulnerable to certain

performance degradation attacks that were described by Amir *et al.* (2011). However, it is important to notice that in our protocol the client pays the high price of generating the signatures, while the replicas only verify them, which is at least an order of magnitude less costly.

3.6.2 Checkpoints and State Transfer

The second important optimization is related with bounding the size of the decision log. In Mod-SMaRt, such log can grow indefinitely, making it inappropriate for real systems. To avoid this behavior we propose the use of checkpoints and state transfer. Checkpoints would be performed periodically in each replica, after some number D of decisions were delivered, and they would request the state from the application, save it in memory or disk, and clear the log.[1] If in the end of a synchronization phase a replica detects a gap between the latest decision of its own log, and the latest decision of the log it chose, it invokes a state transfer protocol. Such protocol would request from the other replicas the state that was saved in their latest checkpoint. Upon the reception of $f + 1$ matching states from different replicas, the protocol would force the application to install the new state, and the protocol would resume execution.

3.6.3 Optimized Synchronization Phase

The third possible optimization is related to leader driven consensus algorithms, which are prone to choose a faulty process as leader. Because of this, when a majority of processes suspect the current leader is faulty, they deterministically elect a new leader. When a new leader is elected, it needs to discover if some value might already been decided by a correct process. This can be done by collecting the consensus' state of $n - f$ processes, computing a safe value based on those states, and forwarding such value to all processes. If the consensus algorithm uses digital signatures to sign the states that each process sends to the leader, the message pattern is the same as the STOPDATA and SYNC messages, as illustrated in Figure 3.4. Therefore, to reduce the number of exchanged messages in a synchronization phase – at the cost of breaking the modularity of standard Mod-SMaRt – the consensus states of

[1] Notice that, on contrary to the PBFT checkpoint protocol (Castro & Liskov, 2002), Mod-SMaRt checkpoints are local operations.

each process could be piggy-backed in the STOPDATA, and the safe value in the SYNC message.

This optimization requires the execution of the consensus primitive to be *stoppable*, and the consensus state of each process to be *returnable*. This could be achieved using a *Stop-Consensus(i)* procedure, that would stop the execution of an instance and return its current state. This procedure would be invoked before transmission of the STOPDATA message and its returning data would be included in that message, alongside the decision log. Furthermore, the consensus primitive would also need to provide a deterministic function *safeValue(states)* that, upon taking a collection of $n - f$ states from distinct processes, returns a safe value or $\perp$ (if no value could be chosen from the collection of states). This predicate would be invoked upon reception of the STOPDATA messages. Finally, upon the reception of the SYNC message, the execution of the consensus primitive would be resumed using a procedure *ResumeConsensus(i,states)*, that would internally invoke the *safeValue(states)* function to obtain a safe value for the next proposal.

3.6.4 Obtaining PBFT from Mod-SMaRt

If we create our VP-Consensus primitive from Byzantine consensus algorithm matching the generalization given in (Lampson, 2001) and employ the optimizations described previously, we can obtain a protocol very similar to PBFT in respect to its message pattern and computational cost. This is what was done in Chapter 4, were we describe our implementation of the Mod-SMaRt protocol adopting the optimizations discussed so far, together with a VP-Consensus implementation based on the Byzantine consensus algorithm by Cachin (2009). The formal specification of this implementation can be found in Appendix B.

3.7 Additional Related Work

The relationship between total order broadcast and consensus for the Byzantine fault model is studied in many papers. Cachin *et al.* (2001) show how to obtain total order broadcast from consensus provided that the latter satisfy the *External Validity* property. Their transformation requires an echo broadcast plus public-key signature, adding thus at least two communication steps (plus the cryptography delay) to the consensus protocol. Correia *et al.* (2006) proposed a similar reduction without relying on public-key signatures, but using a reliable broadcast

and a multi-valued consensus that satisfies a validity property different from Cachin's. The resulting transformation adds at least three communication steps to the consensus protocol in the best case. Milosevic *et al.* (2011) take in consideration many variants of the consensus *validity* property proposed in the literature, and show which of them are sufficient to implement total order broadcast. They also prove that if a consensus primitive offers the *validity* property proposed in (Dolev & Hoch, 2008), then it is possible to obtain a reduction of atomic broadcast to consensus with constant time complexity — which is not the case of the previously mentioned reductions (Cachin *et al.*, 2001; Correia *et al.*, 2006). However, their transformation still requires a reliable broadcast, and thus adds at least three communication steps to the consensus protocol. Doudou *et al.* (2005) show how to implement BFT total order broadcast with a weak interactive consistency (WIC) primitive, in which the decision comprises a vector of proposed values, in a similar way to a vector consensus (Correia *et al.*, 2006). They argue that WIC primitive offers better guarantees than a Byzantine consensus primitive, eliminating the issue of the *validity* property of consensus. The overhead of this transformation is similar to (Cachin *et al.*, 2001): echo broadcast plus public-key signature.

All these works provide reductions from total order broadcast to Byzantine consensus by constructing a protocol stack that does not take into account the implementation of the consensus primitive; they only specify which properties such primitive should offer — in particular, they require some strong variant of the *Validity* property. Mod-SMaRt requires both a specific kind of *Validity* property, as well as a richer interface, as defined by our VP-Consensus abstraction. The result is a transformation that adds at most one communication step to implement total order broadcast, thus matching the number of communication steps of PBFT at the cost of using a gray-box consensus abstraction.

There are many works dedicated to generalize, classify and decompose consensus algorithms. Lampson (2001) proposed an abstract Paxos algorithm, from which several other versions of Paxos can be derived (e.g., Byzantine, classic, and disk paxos). Another unification of Paxos-style protocols is presented in (Li *et al.*, 2007), with the unification reduced to a write-once register that offers a special set of semantics and properties. Implementations of such register are given for different system and failures models. Rütti *et al.* (2010) expands this study to propose a more generic construction than Lampson (2001), and identify three classes of consensus algorithms. More recently, Maric *et al.* (2015) proposed a more elaborated taxonomy of consensus algorithms based on the mechanisms and design choices adopted across the literature. Cachin (2009) proposes a simple and elegant modular

decomposition of Paxos-like protocols and show how to obtain implementations of consensus tolerating crash or Byzantine faults based in the factored modules. All these works aim to modularize Paxos either for implementing consensus or SMR under various assumptions. Our work, on the other hand, aims at using a special kind of consensus to obtain BFT SMR in a modular, but latency-optimal manner.

Finally, Milosevic *et al.* (2013) presented BFT-Mencius, a BFT SMR protocol that is also modular and built on top of an abstraction called Abortable Timely Announced Broadcast (ATAB). However, whereas Mod-SMaRt and VP-Consensus are designed to obtain a BFT SMR transformation that preserves optimal latency in terms of number of communications steps during normal execution, BFT-Mencius and ATAB's are designed to obtain a performance-oriented criterion known as *bounded-delay* (Amir *et al.*, 2011).

3.8 Concluding Remarks

In this chapter we describe how to bridge the gap between efficient monolithic BFT SMR protocols and modular BFT atomic broadcast algorithms by presenting Mod-SMaRt, a latency- and resiliency-optimal BFT SMR algorithm that achieves modularity using a well-defined consensus primitive. To achieve optimal number of communication steps, we introduce the *Validated and Provable Consensus* abstraction, which can be implemented by making simple modifications on existing consensus protocols. The protocol here presented serves as the foundation for the thesis next chapter, where we describe its implementation in BFT-SMaRt, an open-source BFT SMR library.

4

BFT-SMaRt

In Chapter 3 we presented the thesis' first step towards an efficient and reliable SMR implementation by focusing on an BFT protocol that is both modular and latency-optimal. By contrast, this chapter describes our effort in implementing and maintaining BFT-SMaRt, a robust Java-based BFT SMR library which implements the protocol proposed in Chapter 3. BFT-SMaRt targets not only high-performance in fault-free executions, but also correctness if faulty replicas exhibit arbitrary behavior. Besides its robustness, BFT-SMaRt is the first BFT SMR system to provide efficient and transparent support for durable services (Bessani *et al.*, 2013) and to fully support reconfiguration of the replica set (Aguilera *et al.*, 2010; Lamport *et al.*, 2010).

This chapter is organized as follows: Section 4.2 and 4.3 describe the design of BFT-SMaRt and its implementation, respectively. Section 4.4 gives an overview of the library's API and programing model. Section 4.5 describes an extensive evaluation of our system. Section 4.6 highlights some lessons learned during the development and maintenance of the system. Finally, Section 4.7 presents our concluding remarks.

4.1 SMR Research vs SMR Usage

The last decade and a half has seen an impressive amount of papers on Byzantine Fault-Tolerant BFT SMR (Abd-El-Malek *et al.*, 2005; Amir *et al.*, 2011; Aublin *et al.*, 2015; Behl *et al.*, 2017; Castro & Liskov, 2002; Clement *et al.*, 2009b; Kapitza *et al.*, 2012; Kotla *et al.*, 2009; Veronese *et al.*, 2013, 2009), but almost no practical use of these techniques in real

deployments. Our view of this situation is that the fact that there are no robust-enough implementations of BFT SMR available, only prototypes used for validating novel ideas in papers, makes it quite difficult to use this kind of technique. The general perception is that implementing BFT protocols is too complex and that commission faults are rare and can be normally dealt with simpler techniques like checksums (Correia *et al.*, 2012).

To the best of our knowledge, from all "BFT systems" that appeared on these fifteen years, only the early PBFT (Castro & Liskov, 2002) and the more recent UpRight (Clement *et al.*, 2009a) and Prime (Amir *et al.*, 2011) implement a complete replication system. However, PBFT employs a single-threaded architecture which does not fully exploit modern hardware, UpRight uses two additional layers of servers between clients and replicas (besides presenting a performance an order of magnitude lower than the other systems), and Prime's performance is affected by its several mechanisms for detection and prevention of Byzantine behavior. Furthermore, both PBFT and UpRight are plagued by bugs and are not maintained anymore. Even considering crash-only fault-tolerant (CFT) replication libraries, which are usually based on the many variants of Paxos (Lamport, 1998) or Raft (Ongaro & Ousterhout, 2014), it seems there is still no widely-used robust implementation that can be used for developing dependable services. As a result, every organization that requires such services need to develop its own implementation (e.g., (Chandra *et al.*, 2007)). Therefore, the main contribution of this chapter is to fill a gap in the BFT literature by documenting the implementation of this kind of system, including associate protocols for state transfer and reconfiguration.

4.2 BFT-SMaRt Design

The development of BFT-SMART started at the beginning of 2007 to implement a BFT total order multicast protocol for the replication layer of the DepSpace coordination service (Bessani *et al.*, 2008). In 2009 the implementation was revamped to make it a complete BFT replication library, including features such as checkpoints and state transfer. Nonetheless, it was only during the period of this PhD that the library was substantially improved in terms of performance, functionality, and robustness.

4.2.1 Design Principles

BFT-SMaRt was developed with the following design principles in mind:

Tunable fault model. By default, BFT-SMaRt tolerates *non-malicious Byzantine faults*, a realistic (albeit pessimistic) system model in which messages can be delayed, dropped and even corrupted, while processes can crash or have their state and code corrupted, taking any spurious action as a consequence. All these behaviors have been observed in real systems and components (see Correia *et al.* (2012) for an overview). We believe this is an appropriate fault model for a pragmatical system to support. Besides that, BFT-SMaRt also supports the use of cryptographic signatures for improved tolerance to *malicious Byzantine faults*, or the use of a simplified protocol, similar to Paxos (Lamport, 1998), to tolerate only crashes and message corruptions.[1]

Simplicity. The emphasis on protocol correctness lead us to avoid the use of optimizations that could bring extra complexity both in terms of deployment and coding or add corner cases to the system. For this reason, we avoid techniques that, although promising in terms of performance (e.g., speculation (Kotla *et al.*, 2009) and pipelining (Aublin *et al.*, 2015)) or resource efficiency (e.g., trusted components (Kapitza *et al.*, 2012; Veronese *et al.*, 2013) or IP multicast (Castro & Liskov, 2002; Kotla *et al.*, 2009)), would make our code more difficult to render correct (due to new corner cases) or deploy (due to lack of infrastructure support). This emphasis also made us choose Java instead of C/C++ as the implementation language. In Section 4.5 we show that even with these choices, the performance of BFT-SMaRt is similar or better than some of these optimized SMR implementations.

Modularity. As discussed in Section 3.1, modular protocols tend to be easier to implement and reason about when compared to monolithic protocols such as PBFT, even though they are not optimal in terms of number of communication steps. Therefore, to benefit from the advantages offered by modular protocols while preserving optimal performance in terms of number of communication steps, BFT-SMaRt implements the Mod-SMaRt protocol described in Chapter 3. Besides supporting the same modules of Mod-SMaRt for reliable communication, client requests ordering and consensus, BFT-SMaRt also implements state

[1] Unless stated otherwise, we focus on the BFT setup of the system.

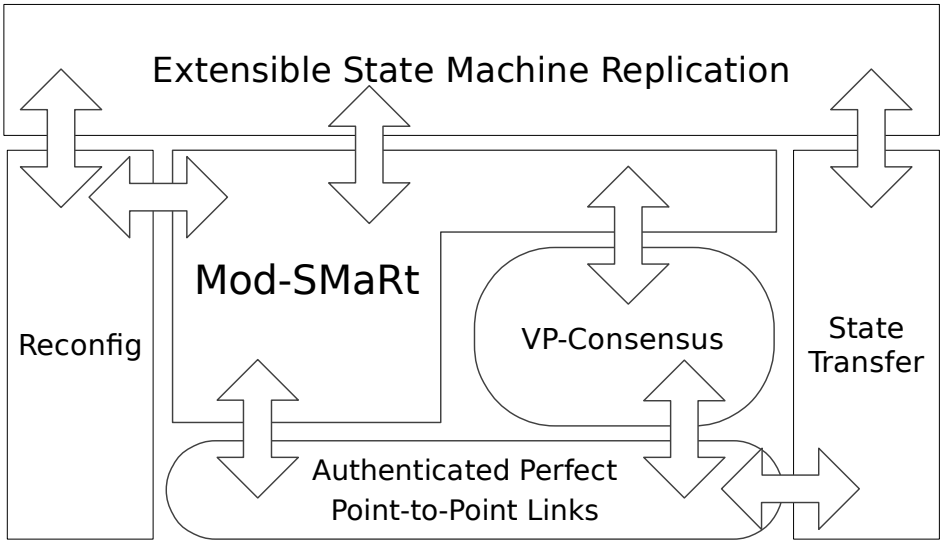


Figure 4.1: The modularity of BFT-SMART.

transfer and reconfiguration modules, which are also separated from the agreement protocol, as show in Figure 4.1.

Simple and Extensible API. Our library encapsulates all the complexity of SMR inside a simple and extensible API that can be used by programmers to implement deterministic services. More precisely, if the service strictly follows the SMR programming model, clients can use a simple *invoke(command)* method to send commands to the replicas, that implement an *execute(command)* method to process the command after it is totally ordered by the Mod-SMaRt protocol. If the application requires advanced features not supported by such basic programming model, these features can be implemented with a set of alternative calls, callbacks or plug-ins both at client- and server-side (e.g., custom voting by the client, reply management and state management, among others).

Multi-core awareness. BFT-SMART takes advantage of ubiquitous multicore architecture of servers to improve some costly processing tasks on the critical path of the protocol. In particular, we make our system throughput scale with the number of hardware threads supported by the replicas, specially when signatures are enabled and more computing power is needed for their verification.

4.2.2 System Model

The system model is similar to what is defined in Section 3.3, but with two key differences: replicas comply to the *fail-recovery* fault model (i.e., processes that are faulty can become correct again) and the replica set is *dynamic*. In order to support fail-recovery and a dynamic set of replicas, the total-order protocol (Section 4.2.3.1) must be augmented with a state transfer and reconfiguration sub-protocols (Sections 4.2.3.2 and 4.2.3.3, respectively).

The system model must also account for replicas that are recovering the state while the remaining replicas keep processing messages. However, replicas that entered this *recovery mode* cannot participate in the total-order protocol, since their state is invalid and/or do not have the necessary context (i.e., do not have information about which replica is the current leader or which consensus instance is currently being processed). Therefore, even though the theoretical lower bound for BFT-SMaRt is also $n \geq 3f + 1$, f must represent the maximum number of replicas that are both faulty *or* entered recovery mode. This means that, if there are t replicas in recovery mode, then there must be at most $f - t$ faulty replicas in the system.[1]

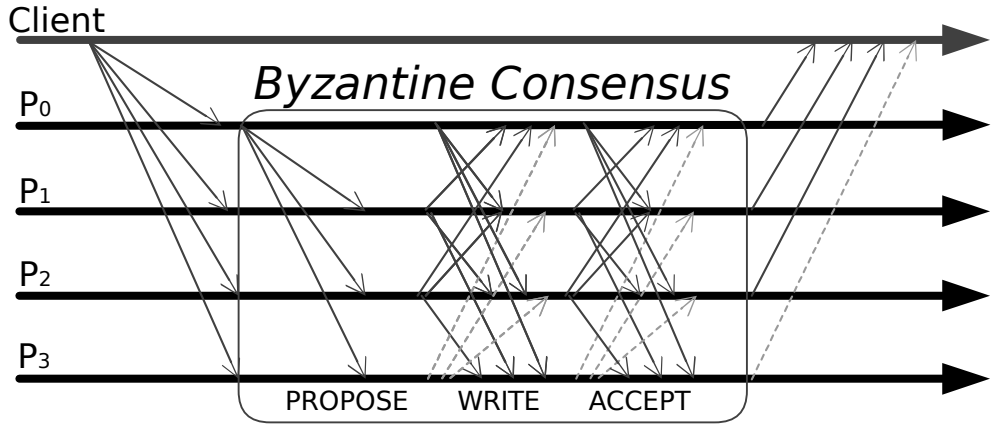
At any instant, only replicas in the *current view cv* of the system are considered by the BFT-SMaRt protocols. The list of servers in *cv* represents the most up-to-date view installed in the system. We denote by $cv.n$ the number of replicas in *cv* and $cv.f < cv.n/3$ the number of replicas in *cv* allowed to fail arbitrarily. When reconfigurations are not being considered we suppress the *cv* prefix.

4.2.3 Core Protocols

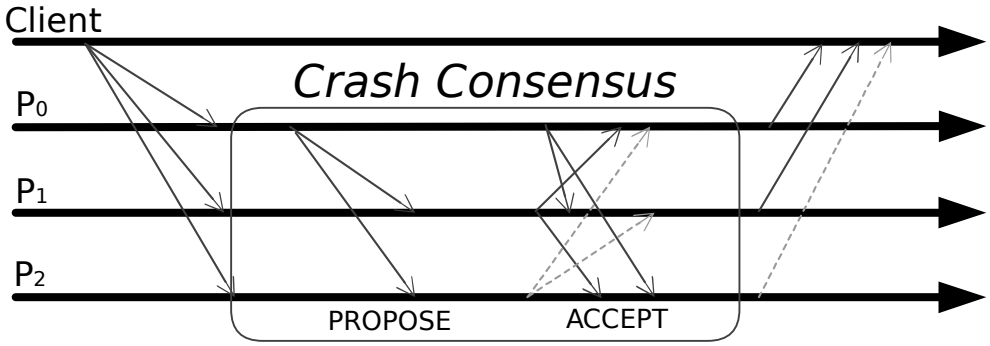
BFT-SMaRt uses a number of protocols for implementing SMR. In this section we give a brief overview of these protocols.

4.2.3.1 Total Order Multicast

Total order multicast is achieved using the Mod-SMaRt protocol presented in Chapter 3 together with a VP-Consensus primitive derived from the Byzantine consensus algorithm described by Cachin (2009) (Appendix B provides details on how VP-Consensus can be



(a) BFT message pattern.



(b) CFT message pattern.

Figure 4.2: BFT-SMART normal phase message patterns.

derived from this algorithm).

Clients send their requests to all replicas in *cv*, and wait for their replies.In the absence of faults and presence of synchrony, BFT-SMART executes in *normal phase*, whose message pattern is illustrated in Figure 4.2a. This phase considers the execution of a sequence of consensus instances, each of them deciding the order of a batch of one or more client requests. Each consensus execution i begins with one of the replicas designated as the leader (initially the replica with the lowest id) proposing some value for the consensus through a *PROPOSE* message. All replicas that receive this message verify if its sender is the current leader, and if the value proposed is valid (i.e., it contains only authenticated requests not yet ordered), they weakly accept the value being proposed, sending a *WRITE* message to other replicas. If

[1] An alternative would be to expand the lower bound to $n \geq 3f + 2k + 1$ replicas (Sousa, 2006), but this is undesirable in practice.

some replica receives $\lceil \frac{n+f+1}{2} \rceil$ *WRITE* messages for the same value, it strongly accepts this value and sends an *ACCEPT* message to other replicas. If some replica receives $\lceil \frac{n+f+1}{2} \rceil$ *ACCEPT* messages for the same value, this value is used as the decision for consensus. The collected *ACCEPT* messages of a consensus instance form a certificate of its decision (i.e., a proof for the VP-Consensus). Therefore, such messages include a MAC vector, to enable the validation of a decision after a leader change. Finally, the decision is logged (either in memory or disk) and the requests in the decided batch are executed in a deterministic order. The normal phase of the protocol is executed in the absence of faults and in the presence of synchrony. When these conditions are not satisfied, the *synchronization phase* might be triggered. During this phase, Mod-SMaRt must ensure three things: (1) a quorum of $n - f$ replicas must have the pending messages that caused the timeouts; (2) correct replicas must exchange logs to converge to the same consensus instance; and (3) a timeout is triggered in this consensus, proposing the same leader at all correct replicas (see Section 3.5.4 for details).

As mentioned before, BFT-SMaRt can also be configured for CFT only. In this case it implements a Paxos-like message pattern Lamport (1998), illustrated in Figure 4.2b. The main differences are that the CFT protocol does not require *WRITE* messages, waits only for $\lceil \frac{n+1}{2} \rceil$ *ACCEPT* messages, and requires a simple majority of non-faulty replicas to preserve correctness.

4.2.3.2 State Transfer

In order to implement a practical SMR system, the replicas should be able to be repaired and reintegrated in the system, without restarting the whole replicated service. Furthermore, the possibility of correlated failures that can bring down more than f replicas of the system at once requires the employment of durability techniques (e.g., the use of stable storage) to recover the whole system in such situations. BFT-SMaRt implements the efficient durability techniques described in (Bessani *et al.*, 2013) to deal with the recovery of replicas or the whole system. In the following we give an overview of such techniques.

By default, replicas store each batch of ordered requests to a (stable) log and, periodically, take snapshots of the application state and store it in stable memory. These two techniques incur a non-negligible performance penalty when disks are used. To mitigate this effect, the

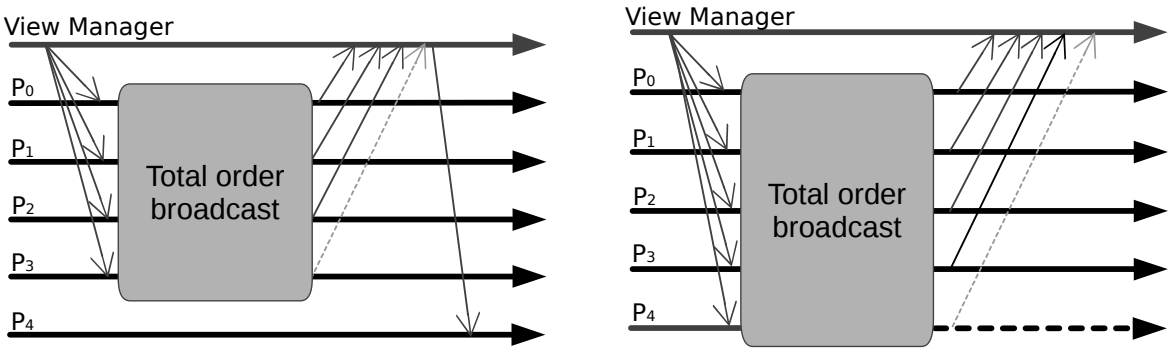
logging of operations can be done in batches and in parallel with their execution while snapshots are taken at different points of the execution in different replicas (Bessani *et al.*, 2013). This behavior is implemented in an well-defined layer between the replication protocol and the application, and it can be changed in accordance with the requirements of the application.

The default state transfer protocol can be triggered either when (1) a replica crashes but is later restarted, (2) a replica detects that it is slower that the others, (3) a synchronization phase is triggered but the log is truncated beyond the point at which the replica could apply the operations, and (4) a replica is added to the system while it is running (see next section). When any of these scenarios are detected, the replica sends a *STATE_REQUEST* message to all the other replicas asking for the application's state. Upon receiving this request, they reply with a *STATE_REPLY* message containing the version of the state that was requested by the replica. Instead of having one replica sending the complete state (checkpoint and log) and others sending cryptographic hashes for validating this state, as is done in PBFT and other systems, we use a partitioning scheme in which one replica sends a checkpoint and the others send parts of the logs (Bessani *et al.*, 2013).

4.2.3.3 Reconfiguration

All previous BFT SMR systems assume a static system that cannot grow or shrink over time. BFT-SMART, on the other hand, provides an additional protocol that enables the system current view cv to be modified at runtime, i.e., replicas can be added or removed without stopping the system. In order to accomplish this, BFT-SMART uses a special type of client named *View Manager*, which is a trusted third party managed only by system administrators. It can also remain off-line, being required only for adding and removing replicas.

Server operation. The reconfiguration protocol (depicted in Figure 4.3) works as follows: the View Manager issues a *signed request* containing a special reconfigure operation to be processed by the Mod-SMaRt algorithm just like any other client operation. Through this operation, the View Manager notifies the system about the IP addresses, ports, and ids of the replicas it wants to add to (or remove from) the system. In the current BFT-SMART implementation, the View Manager can also request the update of the number of failures tolerated in the system ($cv.f$, which is also part of a view). Since these operations are totally ordered (just like client requests), all correct replicas will adopt the same view as the system' current view cv at any given point in the execution of client operations.



(a) JOIN pattern.

(b) LEAVE pattern.

Figure 4.3: BFT-SMaRt reconfiguration message patterns.

A subtle issue with reconfiguration requests is that *ACCEPT* messages exchanged in the consensus in which they are ordered should be signed (instead authenticated using MAC vectors). This happens because such messages are used to build certificates that may be needed in a future synchronization phase (i.e., leader change), and MAC vectors generated in a view cannot always be verified in a posterior view.

Once the View Manager operation is ordered, it is not delivered to the application. Instead, the request signature is verified to assess if it was produced using the view manager private key. If the signature is valid, the system current view cv is updated in accordance with the updates requested in the reconfigure operation. Moreover, the replicas start establishing a secure channel with the new replicas joining the system (or closing channels with the replicas leaving the system). Finally, the replicas reply to the View Manager informing it if the view change succeeded. In the case of replicas joining the system, the View Manager sends a special message to the replicas that are waiting to join the system, informing them that they can start executing in cv (Figure 4.3a). After this point, the joining replicas trigger the state transfer protocol to bring themselves up to date. In case is the a replica is leaving the system, it simply ceases to process Mod-SMaRt messages.

Client operation. In order to support reconfigurations, each client c also needs to handle a current view variable cv_c that stores the current view known by itself. All client operations need to carry cv_c and the replicas reject any operation issued to an old view, replying instead with their current view cv. The client then updates cv_c and restarts its operation, avoiding access to an outdated view.

Like several other reconfigurable systems (Aguilera *et al.*, 2010), to ensure that a slow client c always terminate its operation op, the number of reconfigurations executed concurrently with op must to be finite. This ensures that c will restart op due to reconfigurations a finite number of times, eventually completing it.

The last requirement of a reconfigurable system is that, before accessing the system, a client must obtain the system' current view. This can be done with the use of a directory service (Aguilera *et al.*, 2010; Lorch *et al.*, 2006), for example.

4.2.4 Intrusion Tolerance

Making a BFT replication library tolerate intrusions requires one to deal with several concerns that are not addressed by most BFT protocols (Bessani, 2011). Here we discuss how BFT-SMART deals with some of these concerns.

Previous works showed that the use of public-key signatures on client requests makes it impossible for clients to forge MAC vectors and force leader changes (making the protocol much more resilient against malicious faults) (Amir *et al.*, 2011; Clement *et al.*, 2009b). By default, BFT-SMART does not use public-key signatures[1] other than for establishing shared symmetric keys between replicas and during the synchronization phase. However the system optionally still supports the use of signed requests for avoiding this problem.

These same works also showed that a malicious leader can launch undetectable performance degradation attacks, making the throughput of the system as small as 10% of what would be achieved in fault-free executions. Currently, BFT-SMART does not provide defenses against such attacks. However, the system can be easily extended to support periodic leader changes to limit damage (Clement *et al.*, 2009b). In fact, the codebase of an early version of BFT-SMART was extended to implement a protocol resilient to this kind of attack (Veronese *et al.*, 2009).

Finally, the fact that we developed BFT-SMART in Java makes it easily deployable in different platforms[2] for avoiding single-mode failures, caused by accidental events (e.g., a bug or infrastructure problems) or malicious attacks exploiting common vulnerabilities.

[1] Client requests do not contain MAC vectors also, only point-to-point MACs as provided by the client communication system.

[2] Although we did not support N-versions of the system codebase, we believe supporting the deployment in several platforms is a good compromise solution.

Such compromises on the running platforms can be mitigated by the deployment of replicas in different operating systems (Garcia *et al.*, 2013) or even cloud providers (Vukolić, 2010).

4.3 Implementation

The codebase of BFT-SMaRt contains approximately 15k lines of commented Java code distributed across 139 classes and interfaces. This is significantly less than what was used in similar systems: PBFT (Castro & Liskov, 2002) contains 20k lines of C code and UpRight (Clement *et al.*, 2009a) contains 22k lines of Java code. Even JPaxos (Santos & Schiper, 2013a), the most complete open-source CFT replication library we are aware of, has more than 22k lines of commented Java code.

4.3.1 Building blocks

To achieve modularity, we defined a set of building blocks (or modules) containing the core functionality of BFT-SMaRt. These blocks are divided in three groups: *communication system*, *state machine replication* and *state management*. The first encapsulates everything related to client-to-replica and replica-to-replica communication, including authentication, replay attacks detection, and (re)establishment of communication channels after a failure or reconfiguration. The second implements the core algorithms for establishing total order of requests. The third deals with state management and is described in (Bessani *et al.*, 2013).

4.3.1.1 Communication system

The communication system encapsulates all the code required for receiving requests from clients and messages from other replicas, and sending messages to other processes addressed by their ids. The three main modules are:

- **Client Communication System.** This module deals with the clients that connect, send requests and receive responses from replicas. Given the open-nature of this communication (replicas can serve an unbounded number of clients) we choose the Netty communication framework[1] for implementing client/server communication. The most important requirement of this module is that it should be able to accept and deal with

[1] http://jboss.org/netty

a few thousands of connections efficiently. To do this, the Netty framework uses the `java.nio.Selector` class and a configurable thread pool.

- **Client Manager.** After receiving a request from a client, the replica verifies the authenticity of a request and stores it to be ordered by the replication protocol. For each connected client, this module stores the sequence number of the last request received from this client (to detect replay attacks), the last reply sent to the client (to deal with retransmissions), and maintains a queue containing the requests received but not yet delivered to the service being replicated. The requests to be ordered in a consensus are taken from these queues in a fair way.

- **Server Communication System.** While the replicas accept connections from an unlimited number of clients, as is supported by the client communication system described above, the server communication system implements a closed-group communication model used by the replicas to send messages between themselves. The implementation of this layer was made through "usual" Java sockets, using one thread to send and one thread to receive for each server. One of the key responsibilities of this module is to reestablish the channels between every two replicas after a failure and a recovery.

4.3.1.2 State machine replication

The SMR core was implemented using the simple interface provided by the communication system to access reliable and authenticated point-to-point links. More specifically, BFT-SMART uses six main modules to achieve SMR.

- **Proposer**: this simple module (which contains a single class) implements the role of a proposer, i.e., it defines how to propose a value in a *PROPOSE* message and what a replica should do when it is elected as a new leader.

- **Acceptor**: this module implements the core of the consensus algorithm: *PROPOSE*, *ACCEPT*, and *WRITE* messages are processed and generated (in the case of the latter two) here.

- **Total Order Multicast (TOM)**: this module gets pending messages received by the client communication system and calls the proposer module to start a consensus instance. Additionally, a class of this module is responsible for delivering requests to the service replica and to create and destroy timers for the pending messages of each client.

- **Execution Manager**: this module is closely related to the TOM and is used to manage the execution of consensus instances. It stores information about consensus instances and their rounds as well as who was the leader replica on these rounds. Moreover, the execution manager is responsible to stop and re-start a consensus being executed.

- **Leader Change Manager**: Most of the complex code to deal with leader changes is in this module. Although the rules for validation and verification of executed and pending requests are notoriously hard to understand and implement, the code of this module is sequential (i.e., a set of nested loops) and is not in the protocol critical path. This means that this code does not suffer from concurrency problems and neither needs to be very efficient.

- **Reconfiguration Manager**: The reconfiguration protocol is implemented by this module. To avoid unnecessary modifications in other parts of the codebase, this module provides a consistent view of the group of replicas in the system and the number of tolerated faults.

4.3.2 Staged Message Processing

A key issue when implementing a high-throughput replication middleware is how to break the several tasks of the protocol in an architecture that is robust and efficient at the same time. In the case of BFT SMR there are two additional requirements: the system should deal with hundreds of clients and resist malicious behaviors from both replicas and clients. The architecture for BFT-SMART aims for a balance between high-performance (Behl *et al.*, 2015) and simplicity (Castro & Liskov, 2002) without ignoring multi-core awareness.

Figure 4.4 presents the main architecture with the threads used for staged message processing (Welsh *et al.*, 2001) of the protocol implementation. In this architecture, all threads communicate through *bounded queues* and the figure shows which thread feeds and consumes data from which queues.

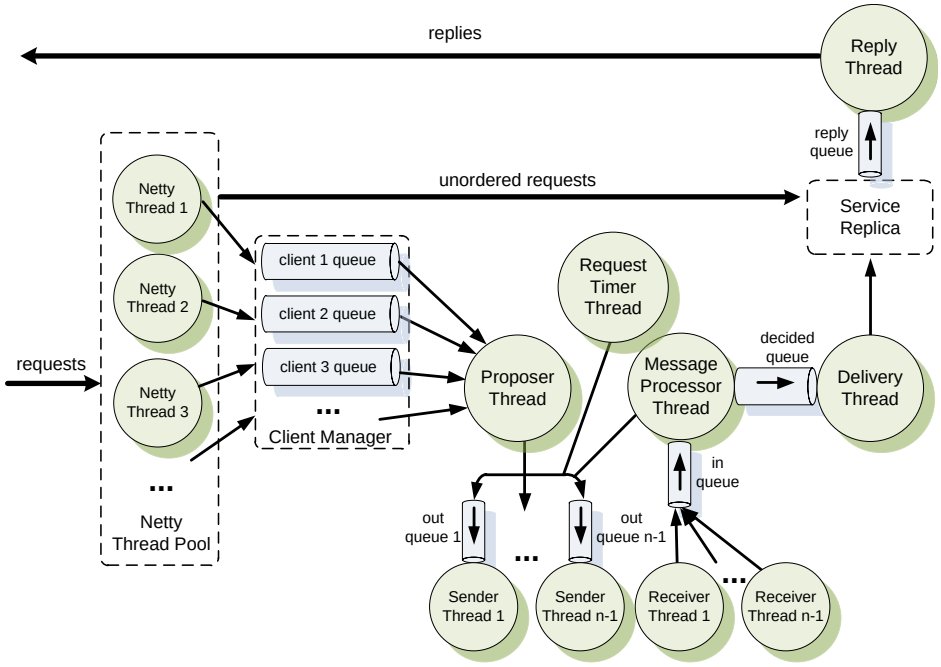


Figure 4.4: BFT-SMART replica staged message processing.

The client requests are received through a thread pool provided by the Netty communication framework. We have implemented a request processor that is instantiated by the framework and executed by different threads as the client load demands. The policy for thread allocation is at most one per client (to ensure FIFO communication between clients and replicas), and we can define the maximum number of threads allowed.

Once a client message is received, it is checked whether it is an ordered or unordered request. Unordered requests are directly delivered to the service implementation. Otherwise, they are delivered to the client manager, that verifies the request integrity and (if validated) adds them to the respective client's pending requests queue. Notice that since client' MACs and signatures (optionally supported) are verified by the *Netty threads*, multi-core and multi-processor machines would naturally exploit their power to achieve high throughput (verifying several client signatures in parallel).

The *proposer thread* waits for three conditions before starting a new instance of the consensus: (i) the replica is the leader for the next consensus; (ii) the previous instance is already finished; and (iii) at least one client (pending requests) queue has messages to be ordered. In a leader replica, the first condition will always be true, and it will propose a

batch of new requests to be ordered as soon as a previous consensus is decided and there are pending messages from clients. Notice the proposal size will contain all pending requests (up to a maximum size, defined in the configuration file), so there is no waiting to fill a batch of certain size before proposing. In non-leader replicas, this thread is always sleeping waiting for condition (i).

Every message m to be sent by one replica to another is put on the *out queue* from which a *sender thread* will get m, serialize it, produce a MAC to be attached to the message and send it using TCP sockets. At the receiver replica, a *receiver thread* for this sender will read m, authenticate it (i.e., validate its MAC), deserialize it and put it on the *in queue*, where all messages received from other replicas are stored in order to be processed.

The *message processor thread* is responsible to process almost all messages of the Mod-SMaRt protocol. This thread gets one message to be processed and verifies if this message consensus is being executed or, in case there is no consensus currently being executed, it belongs to the next one to be started. Otherwise, either the message consensus was already finished and the message is discarded, or its consensus is yet to be executed (e.g., the replica is executing a late consensus) and the message is stored on the *out-of-context queue* to be processed when this future consensus is able to execute. As a side note, it is worth to mention that although the *PROPOSE* message contains the whole batch of messages to be ordered, the *WRITE* and *ACCEPT* messages only contain the cryptographic hash of this batch.

When a consensus is finished on a replica (i.e., the replica received $\lceil \frac{n+f+1}{2} \rceil$ *ACCEPT* messages for the same value), the decision is put on the *decided queue*. The *delivery thread* is responsible for getting decided values (a batch of requests proposed by the leader) from this queue, deserialize all messages from the batch, remove them from the corresponding client pending requests queues and mark this consensus as finalized. After that, the delivery thread invokes the *service replica* to make it execute and log the requests and generate the corresponding replies. When the batch is properly logged and the response is generated by the replica, the service replica adds the reply into the *reply queue*. The *reply thread* is responsible for sending the replies to the clients.

The *request timer thread* is periodically activated to verify if some request remained more than a pre-defined timeout on the pending requests queue. The first time this timer expires for some request, causes this request to be forwarded to the current known leader. The second time this timer expires for some request, the instance currently running of the consensus protocol is stopped and the synchronization phase is started (see Section 4.2.3.1).

The rationale for these timers is the same as explained in Section 3.5.4: a timeout may be caused either by a client that did not send the request to the leader or by a leader that did not ordered the client request. Since typically there are many clients and few replicas, we expect to have much more faults among clients, so we first assume there was a problem with the client and the leader is suspected only if the problem persists.

4.4 API and Programming Model

Two main classes are used to implement a service based on BFT-SMaRt. To instantiate a BFT-SMaRt replica at server side, the `ServiceReplica` class is used, whereas `ServiceProxy` is used at client side for accessing the replicated service. The instantiation of `ServiceReplica` requires the provision of a numeric id (which is mapped to an IP and port through a configuration file) and implementations of an `Executable` – which defines the methods called when the service needs to process a request – and a `Recoverable` – which defines the state management. Usually these two interfaces are implemented by a single class (see bellow). At the client side, the `ServiceProxy` requires only the numeric id of the client. In the following we present a brief overview of the BFT-SMaRt API.

Server-side. The abstract class `DefaultRecoverable` implements the `Executable` and `Recoverable` interfaces considering a simple state transfer manager based on logging and checkpoints. To use this manager, a developer needs to extend the class implementing the following abstract methods:

```
public byte[] appExecuteOrdered(byte[][] cmd, MsgContext[] ctx);
public byte[] appExecuteUnordered(byte[] cmd, MsgContext ctx);
public byte[] getSnapshot();
public void installSnapshot(byte[] state);
```

BFT-SMaRt invokes both the `appExecuteOrdered` and `appExecuteUnordered` methods upon delivering a batch of commands to the application. The former is invoked when clients issue ordered commands, and the latter is invoked for unordered ones (typically read-only operations). In the particular case of `appExecuteOrdered`, BFT-SMaRt delivers a batch of commands previously ordered by the Mod-SMaRt protocol. Both these methods must implement the service code and return replies to be sent to the client. The `cmd` argument represents the serialized command issued by the client, and `ctx` contains

command metadata (e.g., the id of the client, the consensus instance where it was ordered, the latency of the consensus, etc). Additionally, `ctx` also contains a timestamp and a set of nonces which are equal in all replicas. These values are necessary in applications that need to access a local clock or generate random values; they should use these values instead, in order to preserve the determinism property required by SMR (Castro *et al.*, 2003).

Moreover, developers also need to implement `getSnapshot` and `installSnapshot` to create and install serialized snapshots of the application state, respectively. The serialization implemented in `getSnapshot` must be done in a deterministic manner: the snapshot created by a replica r representing state S, must be equal to the snapshot created by replica r' to represent the same state S.

The `DefaultRecoverable` class is usually employed by most BFT-SMaRt-based services. However, an application can use custom implementations of `Executable` and `Recoverable`. For instance, the API provides some specializations of `Executable` with methods to make the service execute one request at time (`DefaultSingleRecoverable`) rather than a batch of commands at once.

The `Recoverable` interface can be used to implement custom state transfer protocols. This class provides a set of callback methods called by the BFT-SMaRt core:

```
public ApplicationState getState(int eid, boolean sendState);
public int setState(ApplicationState state);
public StateManager getStateManager();
```

Developers need to implement `getState` and `setState` to create and define the application state, respectively. These methods require an implementation of `ApplicationState`, an abstract representation of the service state. Moreover, `getStateManager` returns the strategy used to manage state transfer, which can also be implemented by programmers. These features are used to implement the techniques described by Bessani *et al.* (2013).

By default, BFT-SMaRt replies directly to the clients that issued the commands after ordering and executing their requests. However, it is possible to override this procedure by providing a custom `Replier` to the `ServiceReplica`. This can be used (together with asynchronous invocations – see bellow) to implement replicated forwarders (e.g., firewalls, publish-subscribe brokers), where one client (a sender) sends the request to be processed and the "reply" is sent to another client (a receiver) (Garcia *et al.*, 2016).

Client-side. At client side, each instance of `ServiceProxy` represents a single BFT-SMART client with a distinct id. This class provides the following methods to issue commands to the server:

```
public byte[] invokeOrdered(byte[] request);
public byte[] invokeUnordered(byte[] request);
```

These methods are used to issue ordered/unordered commands and require both commands and replies to be serialized into a byte array. In addition, the `AsyncServiceProxy` class can be used to issue both types of commands in a non-blocking manner, i.e., the service proxy will return without waiting for the replicas' replies. This enables programmers to create applications that can resume their execution while the library collects replies in background. To use this feature programmers will have to provide a callback defined by the `ReplyListener` interface to explicitly manage the reception of replies. We used this feature, for example, to implement the client part of a variant of the Byzantium transaction processing protocol (Garcia *et al.*, 2011), which is discussed in Chapter 6.

Finally, the client can also modify how a BFT-SMART client parses replies through the provision of a custom `Comparator` (used to compare server replies) and `Extractor` (used to extract a reply from a set of consistent replies) implementations. This feature is used, for instance, to support the confidentiality mechanisms employed in the DepSpace coordination service, where the servers reply cryptographic shares of a tuple, and the clients verify if they are compatible, extracting the reply by cryptographically combining them (Bessani *et al.*, 2008).

4.5 Evaluation

In this section we present results from BFT-SMART's performance evaluation. These experiments consist of (1) some micro-benchmarks designed to evaluate the library's raw throughput and latency; (2) the comparison of this performance with some competing systems; and (3) an experiment designed to depict the performance's evolution of a small application implemented with BFT-SMART once the system is forced to withstand events like replicas faults, state transfers, and system reconfigurations.

4.5.1 Experimental Setup

Unless stated otherwise, all experiments ran with three (CFT) and four (BFT) replicas hosted in separated machines. The client processes were distributed uniformly across another four machines. Each client machine ran up to eight Java processes, which in turn executed up to fifty threads implementing BFT-SMaRt clients (for a total of up to 1600 clients).

Clients and replicas executed in the Java Runtime environment 1.7.0_21 on Ubuntu 10.04, hosted in Dell PowerEdge R410 servers. Each machine has two quad-core 2.27 GHz Intel Xeon E5520 processor with hyper-threading, i.e., supporting 16 hardware threads, and 32 GB of memory. All machines communicate through an isolated gigabit Ethernet network.

4.5.2 Micro-benchmarks

"Standard" benchmarks. We start by reporting the results we gathered from a set of micro-benchmarks that are commonly used to evaluate SMR systems, and focus on replica throughput and client latency. They consist of a simple client/service implemented over BFT-SMaRt that performs throughput calculations at the server side and latency measurements at the client side. Throughput results were obtained from the leader replica, and latency results from a selected client (always the same). Figure 4.5 presents the results.

The figure illustrates BFT-SMaRt performance in terms of client latency against replica throughput for both BFT and CFT protocols. The standard deviation in all experiments was under 3%. For each protocol we executed four experiments for different request/reply sizes: 0/0, 100/100, 1024/1024 and 4096/4096 bytes. Figure 4.5 shows that for each payload size, the CFT protocol consistently outperforms its BFT counterpart. This difference is due to the smaller number of messages exchanged in the CFT setup, which reflects in less work per client request for the replicas. Furthermore, as the payload size increases, BFT-SMaRt overall performance decreases. This is because (1) the overhead of requests/replies transmission between clients and replicas increases with message size, and (2) since Mod-SMaRt orders requests in batches, the larger is the payload, the bigger (in bytes) the batch becomes, thus increasing its transmission overhead among replicas.

We complement the previous results with Table 4.1, which shows how different payload's combinations affect throughput. This experiment was conducted under a saturated system running 1600 clients using only the BFT protocol. Our results indicate that increasing request's payload generates greater throughput degradation than reply's payload does.

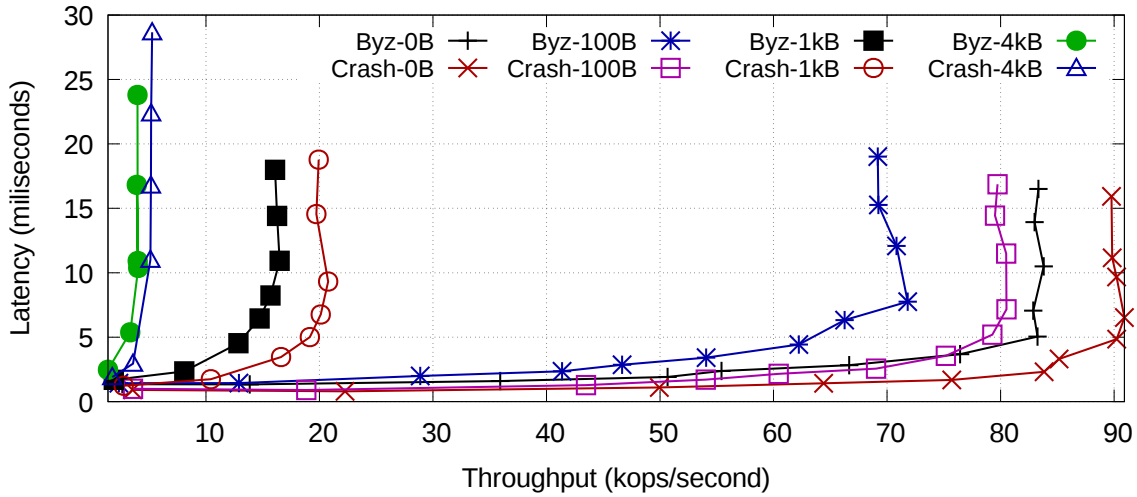


Figure 4.5: Latency vs. throughput configured for $f = 1$.

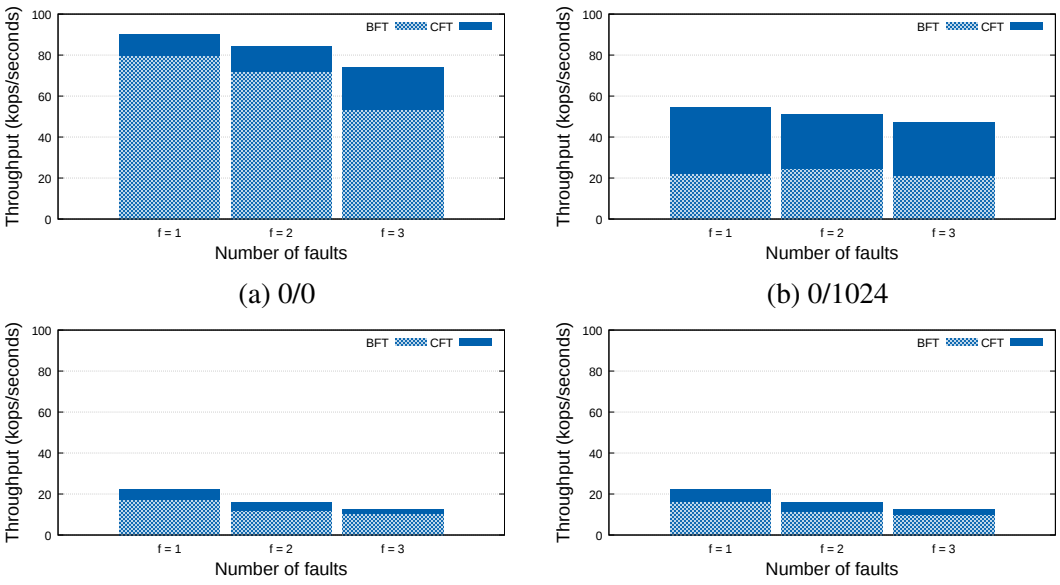
Replies / *Requests*	0 bytes	100 bytes	1024 bytes
0 bytes	83	75	37
100 bytes	79	73	37
1024 bytes	16	16	16

Table 4.1: Throughput in kops/sec for different requests and replies sizes for $f = 1$. Results are given in operations per second.

This can also be explained by the larger batch submitted to the consensus protocol, since request's payload influences its size, whereas reply's does not.

Fault-scalability. Our next experiment consider the impact of the size of the replica group on the peak sustained throughput of the system under different benchmarks. The results are reported in Figure 4.6.

The results show that, for all benchmarks, the performance of BFT-SMART degrades graciously as f increases, both for CFT and BFT setups. In principle, these results contradict the observation that protocols containing all-to-all communication patterns are less scalable as the number of faults tolerated (Abd-El-Malek *et al.*, 2005). This is not the case in BFT-SMART because (1) it exploits the many cores of the replicas (which our machines have plenty) to calculate MACs; (2) only the $n-1$ *PROPOSE* messages of the consensus protocol are large, the other $2n(n-1)$ messages are much smaller and contain only the hash of the



(a) 0/0 (b) 0/1024

(c) 1024/0 (d) 1024/1024

Figure 4.6: Peak sustained throughput of BFT-SMART for CFT ($2f + 1$ replicas) and BFT ($3f + 1$ replicas) considering different workloads and group sizes.

proposed request batch; and (3) we avoid the use of IP multicast, which is know to cause problems with many senders (e.g., multicast storms) (Birman *et al.*, 2009).

Finally, it is also interesting to see that, with relatively big requests (1024 bytes), the difference between BFT and CFT tends to be very small, independently on the number of tolerated faults. Moreover, the performance drops between tolerating 1 to 3 faults is also much smaller with large payloads (both requests and replies).

Mixed workloads. Figure 4.7 reports the results of our experiment considering a mix of read and write requests. In the context of this experiment, the difference between reads and writes is that the former issues small requests (almost-zero size) but gets replies with payload, whereas the latter issues requests with payload but gets replies with almost zero size. This experiment was also conducted under a saturated system running 1600 clients.

We performed the experiment both for the BFT and CFT setups of BFT-SMART, using requests and replies with payloads of 100 and 1024 bytes. Similarly to the previous experiments, the CFT protocol outperforms its BFT counterpart regardless of the ratio of read to write requests by around 5 to 15%. However, the observed behavior of the system regarding

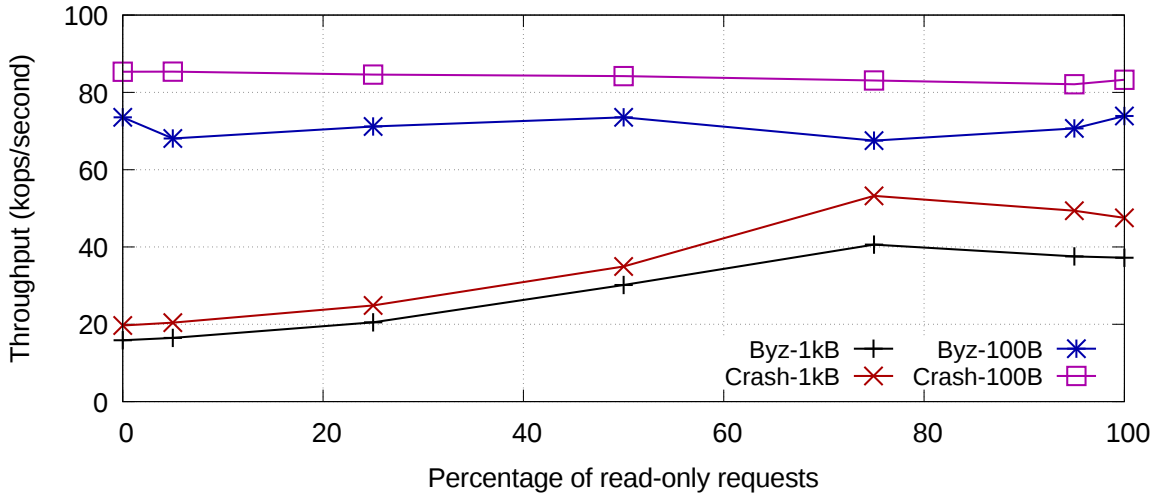


Figure 4.7: Throughput of a saturated system as the ratio of reads to writes increases. Experiment considers $n = 4$ (BFT) and $n = 3$ (CFT).

the throughput differs between the case of 100 and 1024 bytes payloads, with the former clearly benefiting from a larger read/write ratio.

This happens because 1024 bytes requests (a write operation) generate batches much larger than requests with only 100 bytes of payload. This in turn spawns a much greater communication overhead in the consensus protocol. Therefore, as we increase the read to write ratio for payloads of 1024 bytes, the consensus overhead decreases, which in turn improves performance. This happens with up to 75% reads, which has a better throughput than 95%- or 100%-read workloads. This happens for payloads of 1024 bytes because at this point sending the large replies of the read become the contention point of our system. Notice this behavior is much less significant with small payloads.

Signatures and Multi-core Awareness. Our next experiment considers the performance of the system when signatures are enabled, and used for ensuring resilience to malicious clients (Clement *et al.*, 2009b). In this setup a client signs every request to the replicas that first verify its authenticity before ordering it. There are two fundamental service-throughput overheads involved in using 1024-bit RSA signatures. First, the messages are 112 bytes larger than when SHA-1 MACs are used. Second, the replicas need to verify the signatures, which is a relatively costly computational operation.

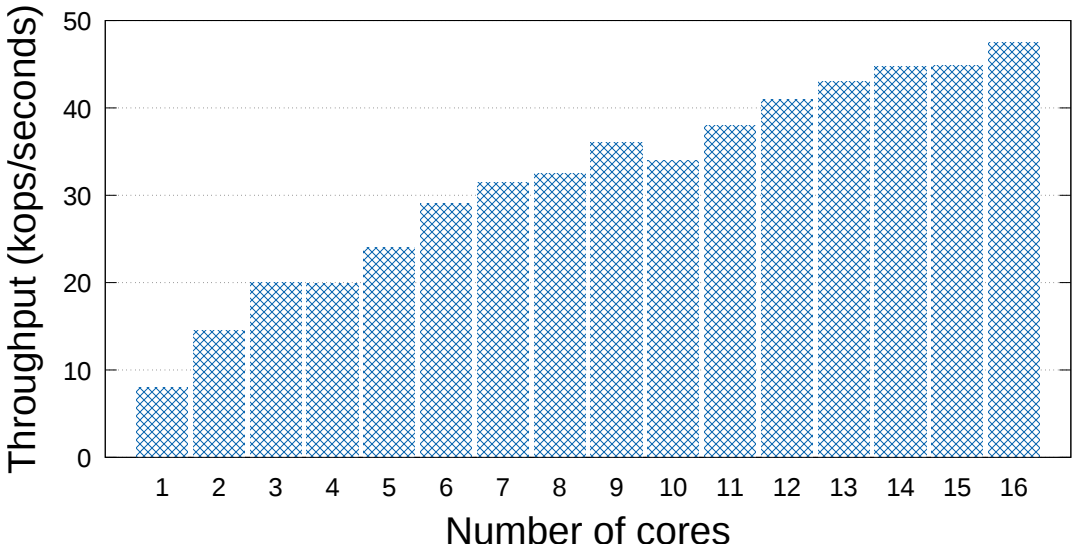


Figure 4.8: Throughput of BFT-SMaRt using 1024-bit RSA signatures for 0/0 payload and $n = 4$ considering different number of hardware threads.

Figure 4.8 shows the throughput of BFT-SMaRt with different number of threads being used for verifying signatures. As the results show, the architecture of BFT-SMaRt exploits the existence of multiple cores (or multiple hardware threads) to scale the throughput of the system. This happens because the signatures are verified by the Netty thread pool, which uses a number of threads proportional to the number of hardware threads in the machine (see Figure 4.4).

Comparison with others. We compared BFT-SMaRt against some representative SMR systems considering the 0/0 benchmark. More precisely, we compared BFT-SMaRt (both in BFT and CFT setups) with PBFT (Castro & Liskov, 2002), UpRight (Clement *et al.*, 2009a) and JPaxos (Santos & Schiper, 2013a) (a CFT replication library). All systems were downloaded from the internet[1] in October 2013, installed and configured to mimic the setup used in their respective papers. In the case of UpRight, we used four machines as servers, three of them with a replica and an ordering server and the last one with only an ordering server. Table 4.2 shows the *peak sustained throughput* obtained for all these systems and the associated number of clients required to achieve this throughput in our environment.

[1]Projects home pages: http://www.pmg.csail.mit.edu/bft/, https://code.google.com/p/upright/ and https://github.com/JPaxos/JPaxos.

System	*Throughput*	*Clients*	*Throughput 200*
BFT-SMART	84	1000	67
PBFT	79	100	66
UpRight	5	600	3
CFT-SMART	91	600	84
JPaxos	63	800	45

Table 4.2: Peak sustained throughput in kops/sec (and associated number of clients used for reaching this value) of different replication libraries for the 0/0 benchmark and $f = 1$. *Throughput 200* reports the throughput obtained by these system with 200 clients.

The results presented in Table 4.2 show that, in our environment, BFT-SMART achieves higher throughput than both PBFT and JPaxos. Even though PBFT reaches its peak throughput with only 10% of the amount of clients required with BFT-SMART, it did not displayed higher throughput with more than 100 clients. We hypothesize that this happens because PBFT is single-threaded, which makes it very efficient with few clients but limits its scalability. Nonetheless, this result is consistent with other reports about PBFT performance (e.g., Correia *et al.* (2012)).

JPaxos displayed a performance lower than what is reported in (Santos & Schiper, 2013a) (around 100 kops/sec). Since we are using the same type of network, the only reason for that is that in the paper they use machines with 24 cores, while our servers support only 16 hardware threads.

As expected, the performance numbers obtained with UpRight were an order of magnitude lower than the others, which is consistent with the values presented by Clement *et al.* (2009a).

Following these results, we sought to get the performance values when the number of clients were the same for all libraries. The table also presents the throughput of the systems with 200 clients for each system.[1]

BFT-SMART displayed again the highest throughput under these conditions. However, notice that PBFT's performance decreased with twice the number of clients. This indicates that the system implementation suffers from some kind of trashing.

[1]The choice of 200 clients was not arbitrary; this is the maximum number of clients supported by PBFT without crashing.

4.5.3 Faults, Reconfigurations, etc.

In this section we present an experiment designed to evaluate the behavior of an application implemented using BFT-SMART, and how it fares against replica's failures, recoveries, and reconfigurations. For this test we use the `BFTMapList` service, an in-memory table storing linked lists associated with each key. This is a simple (but non-trivial) data structure commonly used in practice (e.g., in social network applications).

BFTMapList implementation. `BFTMapList` is an implementation of the `Map` interface from the Java API which uses BFT-SMART to replicate its data in a set of replicas. It can be initialized at the client side providing transparency of the underlying replication mechanism. This is done by invoking BFT-SMART within its implementation. In `BFTMapList`, keys correspond to string objects and values correspond to a list of strings. We implemented the *put*, *remove*, *size* and *containsKey* methods of the aforementioned Java interface. These methods insert/delete a new String/List pair, retrieve the amount of values stored, and check if a given key was already inserted in the data structure. We also implemented an additional method called *putEntry* so that we could directly add new elements to the lists given their associated key.

To evaluate this system, we created client threads that constantly insert new strings of 100 bytes to these lists, but periodically purge them to prevent the lists from growing too large and exhaust memory. Each thread corresponds to one BFT-SMART client.

Results. We sought to observe how BFTMapList performance would evolve upon several events within the system - ranging from replicas faults, leader changes, state transfers and system reconfigurations. For this experiment, BFT-SMART was configured with 4 replicas (with ids ranging from 0 to 3), to tolerate a single Byzantine fault. Our results are depicted in Figure 4.9, presenting throughput values collected from replica 1. We launched 30 clients issuing the *put*, *remove*, *size* and *putEntry* operations over the course of 10 minutes.

As the clients started their execution, the service's throughput increased until all clients were operational around second 10. At second 120 we inserted replica 4 into the service. As we did this, we observed a decrease in throughput. This can be explained by the fact that more replicas demand larger quorums in the consensus protocol and more messages to

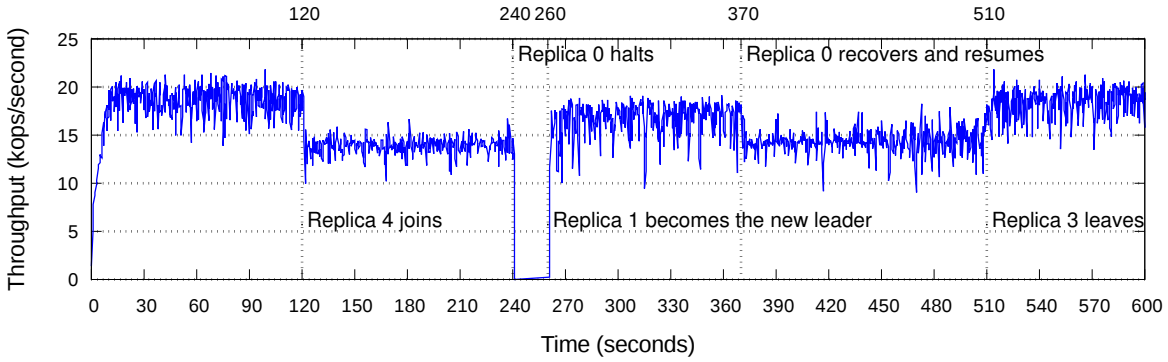


Figure 4.9: Throughput evolution across time and events, for $n = 4$ and $f = 1$.

be processed in each replica. This reconfiguration spawns more message exchanges among replicas, which adds congestion to the network and results in lower performance.

At second 240, we crashed replica 0 (the current consensus' leader). As expected, the throughput dropped to zero during the 20 seconds (twice the timeout value configured in the system) that took the remaining replicas to trigger their timeouts and run Mod-SMaRt's synchronization phase. After this phase was finished, the system resumed execution. Since at this point there are less replicas executing, there are also less messages being exchanged in the system and the throughput was only slightly smaller than in the initial configuration.

At second 370, we restarted replica 0, which resumes normal operation after triggering the state transfer. Upon its recovery, the system goes back to the throughput exhibited before replica 0 had crashed.

At second 510, we removed replica 3, thus setting the quorum size to its original value, albeit with a different set of replicas. Since there is one less replica to handle messages from, we are able to observe the system's original throughput again by the end of the experiment.

4.6 Lessons Learned

More than five years of development and three generations of BFT-SMART gave us important insights about how to implement and maintain high-performance fault-tolerant protocols in Java. In this section we discuss some of the lessons learned on this effort.

4.6.1 Java as a BFT programming language

Despite the fact that the Java technology is used in most application servers and backend services deployed in enterprises, it is a common belief that a high-throughput implementation of a SMR protocol could not be possible in Java (Clement *et al.*, 2009a). We consider that the use of a type-safe language with several nice features (large utility API, no direct memory access, security manager, etc.) that makes the implementation of secure software more feasible is one of the key aspects to be observed when designing a replication library. For this reason, and because of its portability, we choose Java to implement BFT-SMART. However, our experience shows that these advantageous features, when not used carefully, can cripple the performance of a protocol implementation. As an example, we will discuss how object serialization can be a problem.

One of the key optimizations that made our implementation efficient was to avoid Java default serialization in the critical path of the protocol. This was done in two ways: (1) we defined the client-issued commands as byte arrays instead of generic objects, thus removed the serialization and deserialization of this field of the client request from all message transmissions; and (2) we avoid using standard object serialization on client requests, implementing instead a customized method (using data streams instead of object streams). This removed the serialization header from the messages and was specially important for client requests that are put in large quantities on batches to be decided by a consensus.[1]

4.6.2 How to test BFT systems?

Although distributed systems verification and debugging is a lively research area (e.g., Bokor *et al.* (2011); Martins *et al.* (2013)), there are still no tools mature enough to be used. Our approach for testing BFT-SMART is based on the use of JUnit, a popular unit testing tool. In our case we use it in the final automatic test of our build script to run test scripts that (1) setup replicas, (2) run some client accessing the replicated service under test and verify if the results are correct, and (3) kill the replicas in the end. This approach can be automated with the use of fault-injection frameworks and, in fact, one of such tools was recently used to test our system (Martins *et al.*, 2013). Notice that this is black-box testing: the only way

[1] A serialized 0-byte operation request requires 134 bytes with Java default serialization and 22 bytes in our custom serialization.

to observe the system behavior is through the client. Similar approaches are being used in other distributed computing open-source projects like Apache Zookeeper.

Our JUnit-based test framework allows us to easily inject crash-faults on the replicas. However, testing the system against malicious behaviors is much more tricky. The first challenge is to identify the critical malicious behaviors that should be injected on up to f replicas. The second challenge is how to inject the code of the malicious behaviors on these replicas. The first challenge can only be addressed with careful analysis of the protocol being implemented. Disruptive code can be injected to the code using patches, aspect-oriented programming (through crosscutting concerns that can be activated on certain replicas) or simple commented code (which we are currently using). Our pragmatic test approach can be complemented with orthogonal methods such as the Netflix chaos monkey (Bennett & Tseitlin, 2012) to test the system on site.

It is worth to notice that most faulty behaviors can cause bugs that affect the liveness of the protocol, since basic invariants implemented in key parts of the code can ensure safety (e.g., a leader proposing different values to different replicas should cause a leader change, not a disagreement). This means that several recent efforts in verification of safety properties in distributed systems through model checking (e.g., Bokor *et al.* (2011)) does not solve the most difficult problem in our experience: liveness bugs.

Moreover, the fact that the system tolerates arbitrary faults makes it mask some non-deterministic bugs, or Heisenbugs, turning the whole test process even more difficult. For example, an older version of the BFT-SMART communication system losed some messages sporadically when under heavy load. The effect of this was that in certain rare conditions (e.g., when the bug happens in more than f replicas during the same protocol phase) there was a leader change, and the system blocks. We call these bugs *Byzenbugs*, since they are a specific kind of Heisenbugs that happen in BFT systems and that only manifest themselves if they occur in more than f replicas at once. Consequently, these bugs are orders of magnitude more difficult to discover (they are masked) and very complex to reproduce (they seldom happen).

4.6.3 Dealing with heavy loads

When testing BFT-SMART under heavy loads, we found several interesting behaviors that appear when a replication protocol is put under stress. The first one is that there are always

f replicas that stay late in message processing. The reason is that only $n - f$ replicas are needed for the protocol to make progress and naturally f replicas will stay behind. A possible solution for this problem is to make the late replicas stay silent (and not load the faster replicas with late messages that will be discarded) and, when they are needed (e.g., when one of the faster replicas fails), they synchronize themselves with the fast replicas using the state transfer protocol.

Another interesting observation is that, in a switched network under heavy load in which clients communicate with replicas using TCP, spontaneous total order (i.e., client requests reaching all replicas in the same order with high probability) almost never happens. This means that the synchronized communication pattern described in Figure 4.2 does not happen in practice. The main point here is that developers should not assume that client request queues on different replicas will be similar.

The third behavior that commonly happens in several distributed systems is that their throughput tends to drop after some time under heavy load. This behavior is called *trashing* and can be avoided through a careful selection of the data structures[1] used on the protocol implementation and bounding the queues used for threads communication.

4.6.4 Signatures vs. MAC vectors

Castro and Liskov most important performance optimization to make BFT practical was the use of MAC vectors instead of public-key signatures. They solved a technological limitation of that time. When the development of BFT-SMART started, public-key signatures were avoided at all costs due to the fact that the machines we had access at that time created and verified signatures much slower than the machines we used in the experiments described in Section 4.5: a 1024-bit RSA signature creation went from 15 ms to less than 1.7 ms while its verification went from 1 ms to less than 0.09 ms (a $10\times$ improvement). This means that with the machines available today, the problem of avoiding public-key signatures is not so important as it was a decade ago, specially if signature verification can be parallelized (as in our architecture).

[1] For example, data structures that tend to grow with the number of requests being received should process searches in $\log n$ (e.g., using AVL trees) to avoid losing too much performance under heavy load.

4.6.5 Maintenance & Robustness

Our experience with BFT-SMART showed us that implementing a robust BFT system is indeed hard. Several experienced developers that worked in our system mentioned that it was potentially the most complex codebase they had worked on, despite its reasonably modest size. The main observation of these developers was that, at first glance, many parts of the code appear to be unnecessary. The need for these parts was not obvious at first, but they were introduced to deal with bugs that appeared as BFT-SMART was used in more and more projects. This is a consequence of the well-known gap between protocol specifications and descriptions and the code required to implement them efficiently and robustly (Chandra *et al.*, 2007).

We believe BFT-SMART is arguably more robust and efficient than other complete BFT systems (PBFT or UpRight) for a single reason: it is being maintained and constantly improved. Our view is that it is too hard to implement a BFT replication library at once. A more sound strategy is to keep building and improving the system, finding application scenarios and, in the case of academia, looking for opportunities for funding, publication, and student projects as the software evolves.

4.7 Concluding Remarks

This chapter described our effort in building the BFT-SMART state machine replication library. Our contribution with this work is to fill a gap in SMR BFT literature describing how this kind of protocol can be implemented in a safe and efficient way. Our experiments show that the current implementation already provides a very good throughput for both small- and medium-size messages. The BFT-SMART system described here is available as open-source software in the project homepage [1] and, at the time of this writing, there are several groups around the world currently using or modifying our system for their needs. In addition, this library is also fundamental for the next chapter of this thesis, where we explore ways to optimize BFT SMR for geo-replicated scenarios.

[1] http://bft-smart.github.io/library/

5

WHEAT

Chapters 3 and 4 presented the thesis effort towards an efficient and reliable BFT SMR solution both in the theoretical and practical levels. In this chapter we proceed to the next objective of the thesis, where the focus turns to usable BFT SMR in the geo-distributed context. We start by evaluating some representative optimizations proposed in the literature by implementing them in BFT-SMaRt and running the experiments in wide-area environments. Based on this evaluation, we propose WHEAT (WeigHt-Enabled Active replicaTion), a configurable crash and Byzantine fault-tolerant SMR protocol that uses the optimizations we observed as most effective in reducing latency.

The chapter is organized as follows. Section 5.2 presents the results of our empirical evaluation of certain optimizations for decreasing latency. Section 5.3 describes WHEAT, with particular focus on its novel vote assignment scheme and the protocol's evaluation performed at Amazon EC2. Finally, we discuss additional related work in Section 5.4 and present some final remarks in Sectin 5.5.

5.1 From BFT-SMaRt to WHEAT

Many SMR protocols have been proposed for wide area networks (WANs) (e.g., Amir *et al.* (2010); Mao *et al.* (2008); Moraru *et al.* (2013); Veronese *et al.* (2010)). These WAN SMR protocols employ optimizations to reduce latency, usually by decreasing the number of communication steps across the WAN. All these protocols were evaluated in real, emulated or

simulated environments, showing the proposed optimizations were indeed effective in decreasing the protocol latency.

However, even though such evaluations generally use comparable methodologies, they do not use the same experimental environments and codebase across independent works. This lack of a common ground makes it hard to not only compare results across distinct papers, but also to assess which optimizations are actually effective in practice. This is aggravated by the fact that these evaluations tend to compare SMR protocols in an holistic manner, and generally do not compare individual optimizations.

In this chapter we present an extensive evaluations of several latency-related optimizations from the literature (both for local data centers and geo-replication) using the same testbeds, methodology and, codebase. More specifically, we selected optimizations for decreasing the latency of strongly-consistent geo-replicated systems, implemented them in the BFT-SMART and deployed the experiments in the PlanetLab testbed and in the Amazon EC2 cloud.

Unexpected results. During these evaluations, we obtained some unexpected results. The most notorious example is related with the use of multiple leaders – a widely accepted optimization used by several WAN-optimized protocols such as Mencius (Mao *et al.*, 2008) and EPaxos (Moraru *et al.*, 2013). Specifically, our results indicate that this optimization does not bring significant latency reduction just by itself; instead, we observed that using a fixed leader in a fast replica is a more effective (and simpler) strategy to reduce latency. We also found that adding a few more replicas to the system without increasing the size of the quorums required by the protocol may lead to significant latency improvements. These results shed light on which optimizations are really effective for improving the latency of geo-replicated state machines.

New vote assignment schemes. The aforementioned results showcasing the benefit of having extra replicas without necessarily increasing the quorum sizes required by the system led to two novel vote (weight) assignment schemes designed to preserve (CFT and BFT) SMR protocol correctness while also allowing the emergence of quorums of variable size. By allowing quorums of different sizes, it is possible to avoid the need of accessing a majority of replicas – a requirement of many SMR protocols. We introduce two vote assignment schemes (for CFT and BFT SMR) and show that they enable the formation of safe and

minimal quorums without endangering the consistency and availability of the underlying quorum system (Malkhi & Reiter, 1998). To the best of our knowledge, this is the first work that incorporates the idea of assigning different votes for different replicas (i.e., weighted replication) (Garcia-Molina & Barbara, 1985; Gifford, 1979; Pâris, 1986) in replicated state machines.

Weight-Enabled active replication. The results obtained from the experiments and creation of the vote assignment schemes spawned the design, implementation and evaluation of WHEAT (WeigHt-Enabled Active replicaTion), a WAN-optimized SMR protocol developed by extending BFT-SMART with the most effective optimizations (according to our experiments) and our vote assignment schemes. The evaluation of WHEAT – conducted in Amazon EC2 – shows that this protocol could outperform BFT-SMART by up to 56% in terms of latency. To the best of our knowledge, WHEAT is the first SMR protocol that is both optimized for geo-replication and capable of withstanding general Byzantine faults; Mencius (Mao *et al.*, 2008) and EPaxos (Moraru *et al.*, 2013) tolerate only crash faults while BFT protocols like EBAWA (Veronese *et al.*, 2010) or Steward (Amir *et al.*, 2010) requires either each replica to have a trusted component that can only fail by crash, or only tolerate Byzantine faults within a site (i.e., do not tolerate compromised sites), respectively.

5.2 Experiments

In this section we present the experiments conducted to assess the effectiveness of certain optimizations proposed for SMR in wide area networks (Castro & Liskov, 2002; Kotla *et al.*, 2009; Lamport, 1998; Mao *et al.*, 2008; Moraru *et al.*, 2013; Veronese *et al.*, 2010; Zielinski, 2004) and quorum systems (Gifford, 1979; Pâris, 1986). More precisely, we evaluated the following hypotheses related with such optimizations:

1. Fewer communication steps reduces latency (Section 5.2.2);
2. Clients that wait for fewer replies experience lower latency (Section 5.2.3);
3. Smaller quorums can reduce latency (Section 5.2.4);
4. Clients close to the leader experience lower latency than other clients (Section 5.2.5).

Before presenting our results, we describe some general aspects of our methodology.

5.2.1 Methodology

The considered hypotheses were evaluated by implementing the associated optimizations in BFT-SMaRt's code and executing it simultaneously with the original protocol. Our experiments focus on measuring latency instead of throughput, in particular the median and 90th percentile latency perceived by clients. This is due to the fact that throughput can be effectively improved by adding more resources (CPU, memory, faster disks) to replicas or by using better links, whereas geo-replication latency will always be affected by the speed of light limit and perturbations caused by bandwidth sharing. Furthermore, most practical geo-replication works stress that link latencies and variability are the main issues geo-replicated systems have to deal with (e.g., Corbett et. al (2013); Mao *et al.* (2008)).

During the experiments, clients were equally distributed across all hosts, i.e., a BFT-SMaRt replica and a BFT-SMaRt client were deployed at each host that executed the protocol. Similarly to other works (e.g., Hunt *et al.* (2010); Moraru *et al.* (2013)), each client invokes 1 kbyte requests and receives 1 kbyte replies from the replicas that run a *null* service. Requests were sent to the replicas every 2 seconds, and each client writes its observed latency into a log file. This setup enabled us to retrieve results that are gathered under similar network conditions without saturating the resources (CPU and memory) of the hosts used.

The experiments in which we evaluated optimizations to the SMR protocol were conducted mostly in PlanetLab.[1] This testbed is known for displaying unpredictable latency spikes and highly loaded nodes (Duarte *et al.*, 2010; Warns *et al.*, 2008). These conditions allow us to evaluate the optimizations within unfavourable conditions.

Since our experiments are designed to evaluate solely the client latency in fault-free executions, we only report executions in which all hosts were online. However, since PlanetLab's host are regularly restarted and sometimes become unreachable, we could seldom execute each experiment during the same amount of time. Therefore, we had to launch multiple executions for the same experiment, so that within each execution there would be a period in which all hosts were online. In any case, every experiment reported in this section considers at least 24 hours of measurements.

All experiments were configured to tolerate a single faulty replica. Each experiment was executed using between three to five hosts spread through Europe. The unavailability of

[1] http://www.planet-lab.org.

Country	City	Hostname
Poland	Wroclaw	planetlab1.ci.pwr.wroc.pl
England	London	planetlab-1.imperial.ac.uk
Spain	Madrid	planetlab2.dit.upm.es
Germany	Munich	planetlab2.lkn.ei.tum.de
Portugal	Aveiro	planet1.servers.ua.pt
Norway	Oslo	planetlab1.ifi.uio.no
France	Nancy	host4-plb.loria.fr
Finland	Helsinki	planetlab-1.research.netlab.hut.fi
Italy	Rome	planet-lab-node1.netgroup.uniroma2.it

Table 5.1: Hosts used in PlanetLab experiments

nodes already mentioned led us to use a total of eight hosts through all experiments (see Table 5.1).

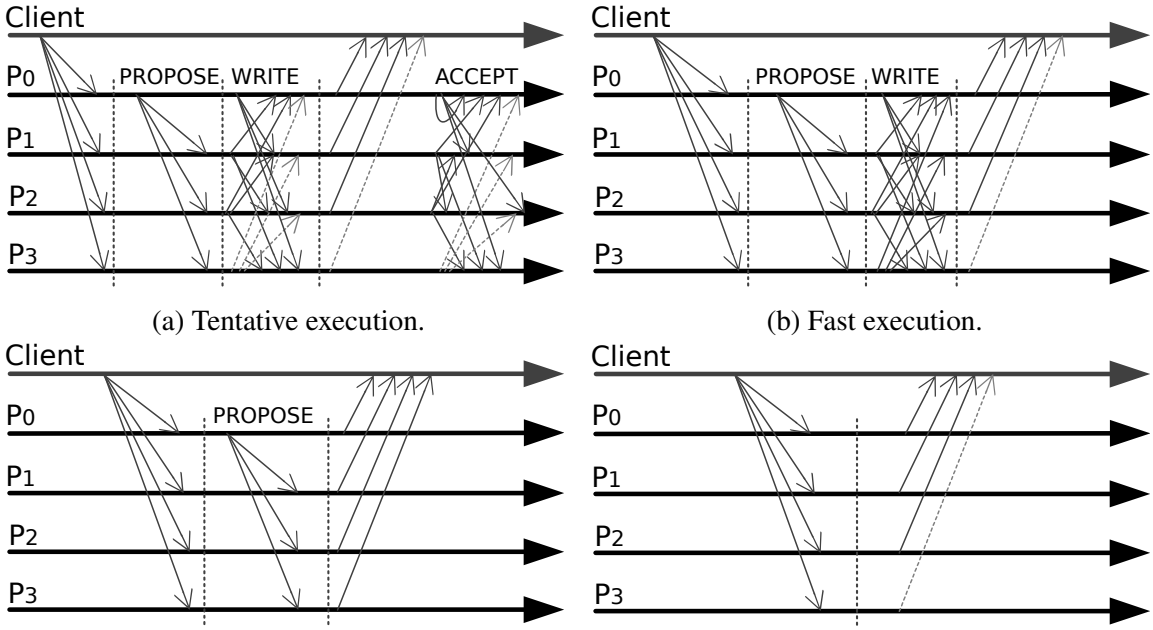
To validate our results in a global scale, two of the experiments were executed on Amazon EC2,[1] using *t1.micro* instances distributed among five different regions. We used the same methodology described for the PlanetLab experiments.

5.2.2 Number of Communication Steps

The purpose of our first experiment is to observe how the client latency is affected by the number of communication steps performed by the SMR protocol. More precisely, we wanted to observe how efficient *read-only*, *tentative*, *speculative* and *fast* executions are in a WAN. The first two optimizations are proposed in PBFT (Castro & Liskov, 2002), whereas the other two optimization are used by Zyzzyva (Kotla *et al.*, 2009) and Paxos at War (Zielinski, 2004), respectively. Since these optimizations target Byzantine-resilient protocols, we only evaluate them in BFT mode.

The message pattern for each of these optimizations is illustrated in Figure 5.1. Figure 5.1a displays the message pattern for tentative executions. This optimization consists of delivering client requests right after finishing the *WRITE* phase, thus executing the *ACCEPT* phase asynchronously. This optimization comes at the cost of (1) potentially needing to perform a rollback on the application state if there is a leader change, and (2) forcing clients to wait for $\lceil \frac{n+f+1}{2} \rceil$ messages from replicas (instead of $f + 1$) (Castro & Liskov, 2002).

[1] http://aws.amazon.com/ec2/.



(a) Tentative execution.

(b) Fast execution.

(c) Speculative execution.

(d) Read-only execution.

Figure 5.1: Evaluated message patterns.

Figure 5.1b displays the message pattern for fast executions. This optimization consists of delivering client requests right after gathering $\lceil \frac{n+3f+1}{2} \rceil$ *WRITE* messages (before the *ACCEPT* phase finishes). If such amount of *WRITE* messages arrive fast enough, the protocol can safely bypass the *ACCEPT* phase. Figure 5.1c displays the message pattern for speculative executions. This optimization enables the protocol to finish executions directly after the *PROPOSE* message is received in the replicas, as long as the clients are able to gather replies from all the replicas within a pre-established time window. If the clients are not able to gather all the replies within such time window, at least one additional round-trip message exchange is required to commit the requests. Figure 5.1d displays the message pattern for read-only executions. This optimization enables clients to obtain a response from the service in two communication steps. However, it can only be used to read the state from the service. Similarly to tentative executions, this optimization also demands that clients gather $\lceil \frac{n+f+1}{2} \rceil$ messages from replicas, *even for non-read-only operations*, to ensure linearizability (Castro & Liskov, 2002).

Setting: Three variants of BFT-SMART were created to evaluate fast, tentative and speculative executions (read-only executions were already supported as unordered requests).

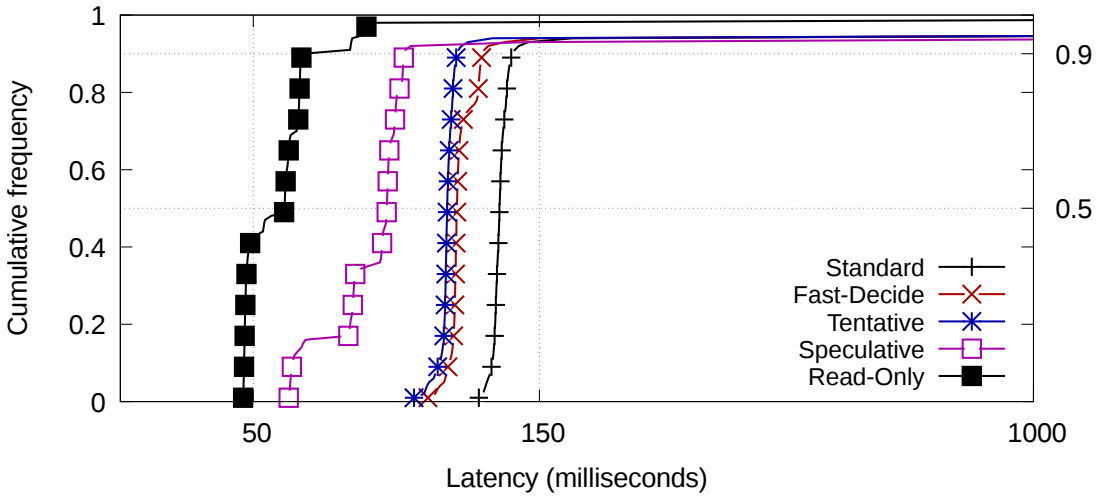


Figure 5.2: Cumulative frequency distribution of latencies for each type of execution.

Client Location	Execution Type				
	Standard	Read-only	Speculative	Tentative	Fast
Nancy (L)	129/135	49/50	58/88	105/109	109/121
Wroclaw	129/136	48/60	73/88	105/110	109/121
Helsinki	129/134	57/73	83/89	105/109	109/120
Rome	129/136	60/77	86/89	105/110	109/121
Overall	129/135	56/61	83/89	105/109	109/121
Improvement	-	**56%/55%**	**36%/34%**	**19%/19%**	**16%/10%**

Table 5.2: Client latencies' 50th/90th percentile (milliseconds) for each type of execution.

This experiment was deployed in Nancy (leader), Wroclaw, Helsinki and Rome.

Results: Figure 5.2 depicts the overall cumulative frequency distribution for the latency observed by the clients for each type of execution. The median and 90th percentile for each client is presented in Table 5.2. All evaluated optimizations exhibited latency reduction across all clients, with read-only executions finishing the protocol execution significantly faster than any of the other optimizations (i.e., 90th percentile latency from 43% to 63% smaller than in standard executions and an overall improvement of 55%). Moreover, speculative executions also displayed significant latency reduction, reaching an overall 90th percentile (resp. median) latency 34% lower (resp. 36% lower) than standard execution.

In the same way, tentative and fast executions also manage to reach a lower median and 90th percentile than standard executions, albeit with more modest differences. Furthermore,

whereas fast executions displayed a latency decrease of about 10%, tentative executions managed to reduce latency by almost 20% (when compared to standard executions).

Main conclusion: The lowest latency displayed by read-only executions were to be expected, since they bypass all three communications steps executed between sending requests and gathering replies. Since speculative executions require the *PROPOSE* phase, they show higher latency than read-only executions. Moreover, the advantage of tentative executions over fast executions can be explained by the fact that the latter require gathering *WRITE* messages from all four replicas, whereas the former only need it from three.

5.2.3 Number of Replies

In this experiment we intended to observe how the amount of replies required by clients affects the operation latency. By default, BFT-SMaRt clients wait for $\lceil \frac{n+f+1}{2} \rceil$ (BFT) or $\lceil \frac{n+1}{2} \rceil$ (CFT) replies from replicas to ensure linearizability. However, this number of replies is required due to the use of read-only executions (Castro & Liskov, 2002): if this optimization were not supported, $f + 1$ matching replies (BFT) or 1 (CFT) reply would suffice.

Setting: a variant of BFT-SMaRt client was developed to wait only for $f + 1$ (BFT) or 1 (CFT) replies, satisfying thus only sequential consistency (similarly to Zookeeper (Hunt *et al.*, 2010)) if the read-only optimization is employed. This experiment was deployed on PlanetLab across hosts located in Nancy (leader), Wroclaw, Helsinki and Rome. The clients from the modified version of BFT-SMaRt waited for two out of four replica replies (or one out of three in CFT), while the original version waited for the usual three out of four (two out of three in CFT). CFT experiments did not require the Helsinki's host.

Results: Figure 5.3 shows the cumulative frequency distribution of the latencies observed by clients. The values for the median and 90th percentile latency for each client are shown in Table 5.3. It can be observed that both the original and modified protocols present very similar performance in BFT mode. On the other hand, the optimization was quite effective in the CFT mode. For the 90th percentile, this optimization showed an improvement from 8% to 11% in BFT mode and from 26% to 36% in CFT mode.

Main conclusion: The lower latency displayed when the protocol requires less replies was to be expected, but such reduction was more significant in CFT mode. This can be explained by the fact that the BFT mode employed one more replica and required one more reply when compared to CFT.

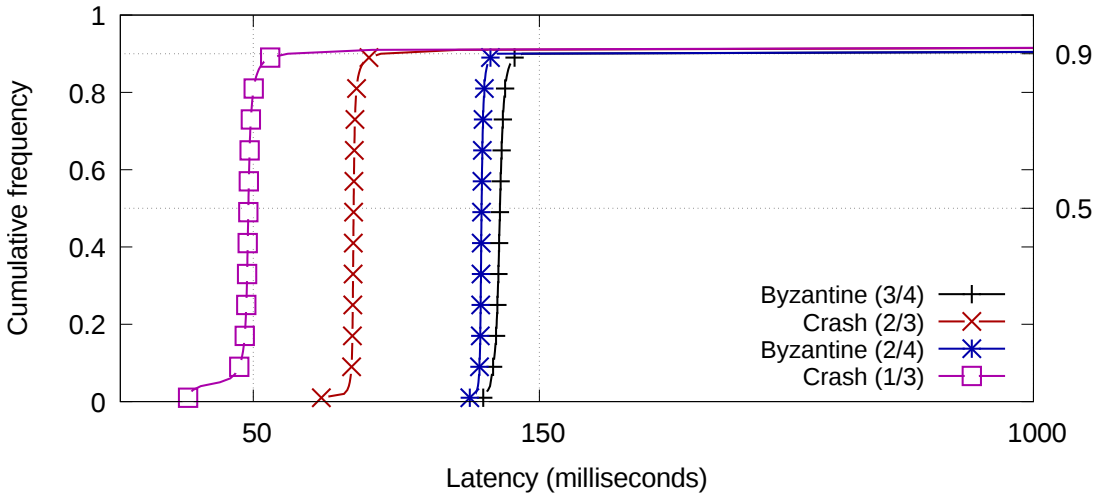


Figure 5.3: Cumulative frequency distribution of latencies for different numbers of replies.

Client Location	Number of replies			
	BFT		CFT	
	3/4	2/4	2/3	1/3
Nancy (L)	129/138	120/127	74/85	49/61
Wroclaw	129/139	120/125	73/80	49/51
Rome	129/143	120/129	73/80	49/59
Helsinki	129/142	120/126	-	-
Overall	129/140	120/127	73/81	49/57
Improvement	-	**7%/9%**	-	**33%/30%**

Table 5.3: Client latencies' 50th/90th percentile (milliseconds) for different numbers of replies.

5.2.4 Quorum Size

This experiment is motivated by the works of Gifford (1979) and Pâris (1986), which use voting schemes with additional hosts to improve the availability of quorum protocols. As described in Section 4.2.3.1, BFT-SMART's clients and replicas always wait for $\lceil \frac{n+f+1}{2} \rceil$ messages from other replicas to advance to the next communication step (or $\lceil \frac{n+1}{2} \rceil$ in CFT mode). More precisely, BFT-SMART waits for *dissemination Byzantine quorums* (Malkhi & Reiter, 1998) if operating in BFT mode and *majority quorums* (Garcia-Molina & Barbara,

1985) if operating in CFT mode. During this experiment, we enable the system to make progress without waiting for the aforementioned quorum types if spare replicas are present. Notice that this optimization, which is not employed in any SMR protocol, might lead to safety violations (discussed below).

Setting: We modified BFT-SMART to make replicas wait for only $2f + 1$ (resp. $f + 1$) messages in each phase of the BFT (resp. CFT) protocol, independently from the total number of replicas n.[1] This experiment was deployed on PlanetLab hosts located in Aveiro (leader), London, Oslo, Munich and Madrid. The original BFT-SMART was configured to execute across four replicas (three in CFT mode) and the modified version was configured to execute in five (four in CFT mode). The extra replica needed for executing the modified version was placed in Madrid, both for BFT and CFT mode. Experiments for CFT mode did not require the use of Munich's host. Since the modified version waits only for three out of five (3/5) messages (or 2/4 messages in CFT mode), both versions of BFT-SMART will wait for the same number of messages, even though the optimized versions use one additional replica.

Results: Figure 5.4 plots the cumulative frequency distribution of latencies for all clients considering the four SMR protocol variants. The median and 90th percentile latency observed by each client for both BFT and CFT modes are reported in Table 5.4. The results show that the modified protocols – which used one extra replica – exhibited lower latency than the original protocols. This difference is more discernible in the CFT mode for two reasons. First, the ratio between the quorum size and the number of replicas (2/4) is smaller than the BFT case (3/5). Second, it did not use London's host (which observed a much worse 90th percentile latency than others). It can be observed that in the 90th percentile, the optimizations showed an improvement of 12%-17% in the BFT mode and 4%-72% in CFT mode, depending on the location of clients.

Main conclusion: The modified version BFT-SMART was able to experience lower latency because it was given more choice: since both versions still waited for the same number of messages in each communication step, the slowest replica was replaced by the extra replica hosted in Madrid, thus decreasing the observed latency of the modified version. In normal protocols this benefit would be smaller, since the quorum size would normally increase with n.

[1] If the original BFT-SMART were deployed in five hosts, the quorums would be comprised of four hosts (in the case of BFT mode).

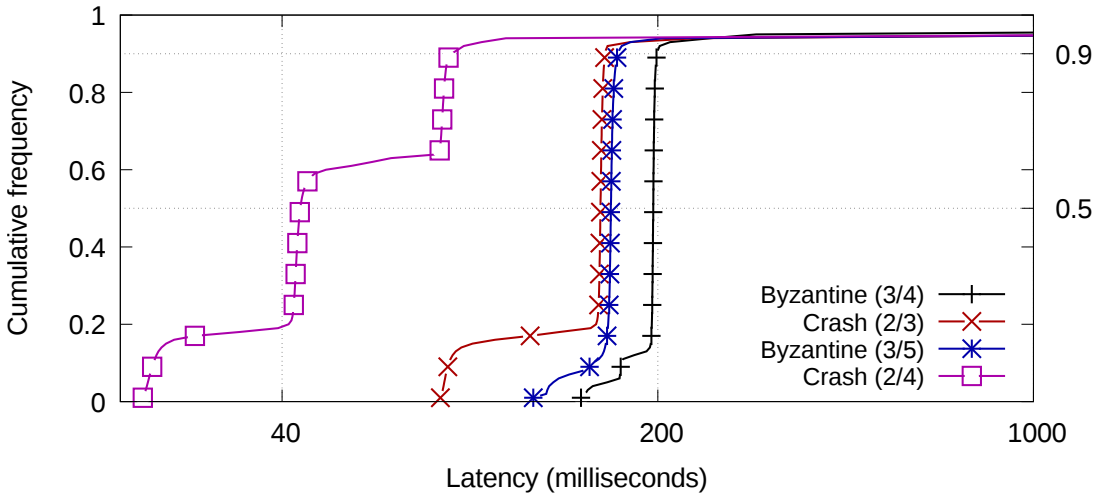


Figure 5.4: Cumulative frequency distribution of latencies with different quorum sizes.

Client Location	Quorum Size			
	BFT		CFT	
	3/4	3/5	2/3	2/4
Aveiro (L)	196/202	163/177	156/161	27/45
London	156/6469	139/5639	85/5856	67/5650
Oslo	196/200	164/171	157/160	80/98
Munich	196/199	164/166	-	-
Madrid	-	164/170	-	42/57
Overall	196/199	164/169	156/160	43/82
Improvement	-	**16%/15%**	-	**72%/49%**

Table 5.4: Client latencies' 50th/90th percentile (milliseconds) with different quorum sizes.

Even though usage of additional replicas reduces latency, this optimization cannot be directly applied to existing protocols without impairing correctness. Limiting the amount of messages to $2f + 1$ (or $f + 1$) regardless of the total number of replicas n does not guarantee formation of intersecting quorums, which are required to ensure safety in both BFT and CFT modes (Castro & Liskov, 2002; Lamport, 1998). For example, in the CFT mode, our setup of $n = 4$ and $f = 1$ did not ensure majority quorums, which can lead to safety violations. To preserve correctness, it is necessary to force any combination of $2f + 1$ (or $f + 1$) replicas to intersect in at least one correct server. Section 5.3.2 presents a mechanism that ensures this property and allows the use of this optimization in SMR systems.

5.2.5 Leader Location

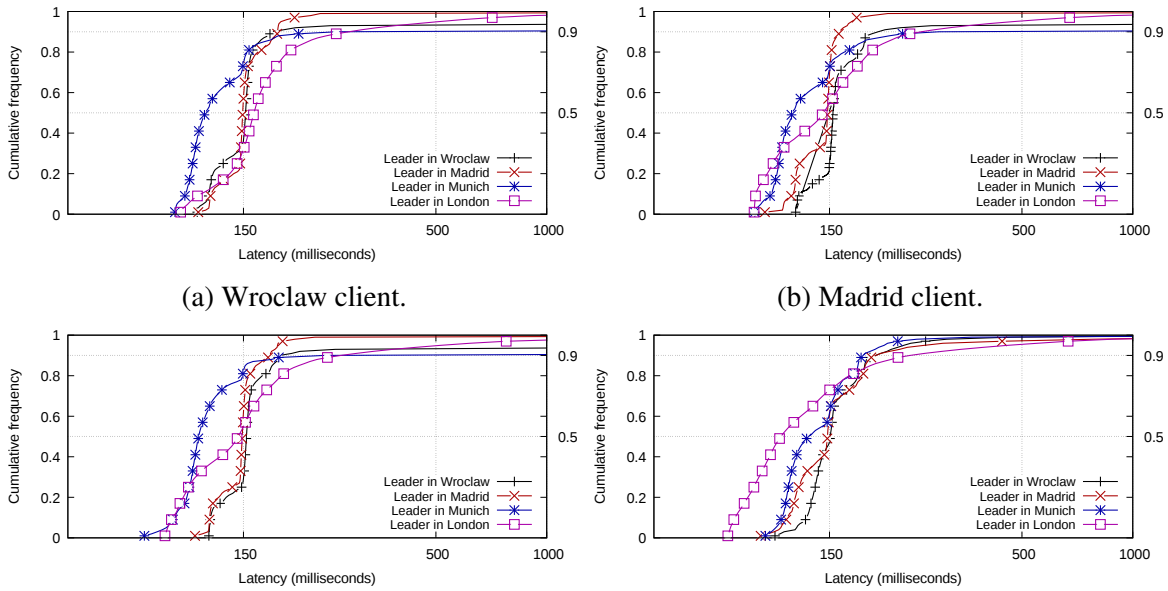
The goal of our last experiment is to observe how much the leader's location can affect the client latency. This experiment is motivated by the fact that Mencius (Mao *et al.*, 2008), EBAWA (Veronese *et al.*, 2010) and EPaxos (Moraru *et al.*, 2013) use different techniques to make each client use its closest (or co-located) replica as the leader for its operations. The rationale behind these techniques is to make client-leader communication faster, bringing down the end-to-end SMR latency.

Setting: We deployed BFT-SMART in PlanetLab and conducted several experiments considering different replicas assuming the role of the leader. The hosts used were located in Wroclaw, Madrid, Munich and London (not used in CFT mode). Moreover, the experiment was repeated across Amazon EC2, using replicas in Ireland, Oregon, São Paulo and Sydney regions (Sydney was only used in BFT mode).

Results: Figures 5.5 and 5.6 depicts the cumulative frequency distribution of latencies gathered by each client, for BFT and CFT modes, respectively. The median and 90th percentile observed by each client for different leader locations is presented in Table 5.5. The highlighted values represent the latency observed by clients in the same location as the leader for that particular experiment.

Before launching this experiment, we expected that, for any client, its latency would be the lowest when its co-located replica were the protocol's leader. However, as can be seen in Figures 5.5 and 5.6, the cumulative frequency distribution of the latencies does not change significantly when the leader location changes. In particular, the 90th percentile latency is, in general, lower when the leader was either in Madrid or Wroclaw, as seen in Table 5.5.

Since these results appeared to contradict the intuition of (Mao *et al.*, 2008; Moraru *et al.*, 2013; Veronese *et al.*, 2010), we decided to repeat this experiment in Amazon EC2, to find if this phenomenon is due to our choice of testbed. Figures 5.7 and 5.8 depicts the cumulative frequency distribution of latencies gathered by Amazon EC2's clients, for BFT and CFT modes, respectively. As with the PlanetLab results, the cumulative frequency distribution of the latency observed by the different clients do not present any significant change as we change the leader location.



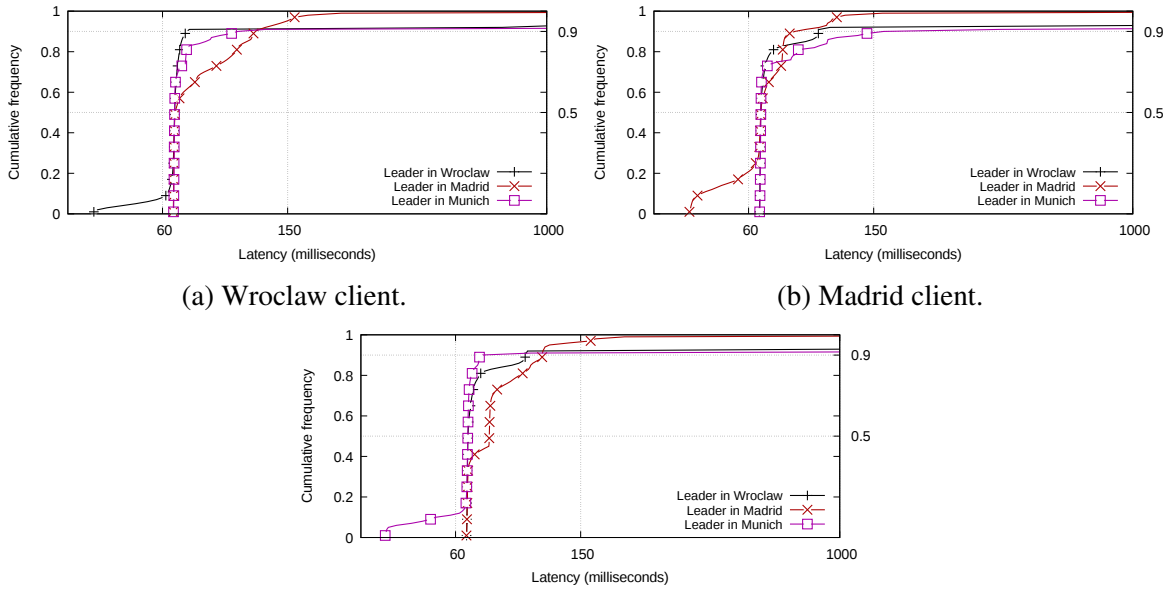
(a) Wroclaw client.

(b) Madrid client.

(c) Munich client.

(d) London client.

Figure 5.5: Cumulative frequency distribution of latencies observed by each client when the leader is placed across PlanetLab hosts (BFT mode).

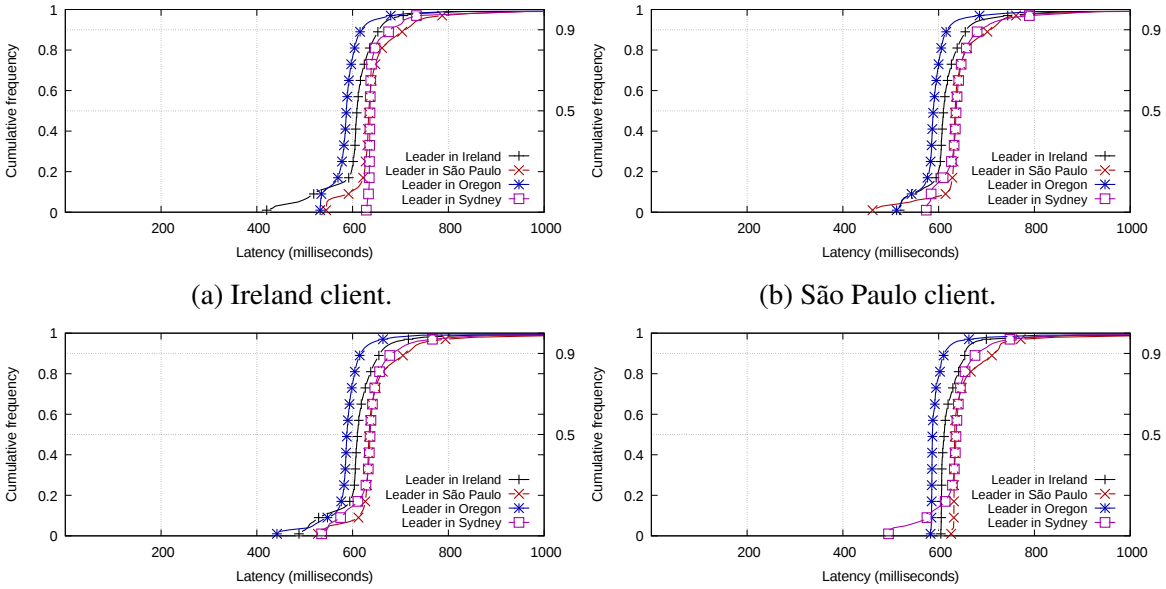


(a) Wroclaw client.

(b) Madrid client.

(c) Munich client.

Figure 5.6: Cumulative frequency distribution of latencies observed by each client when the leader is placed across PlanetLab hosts (CFT mode).



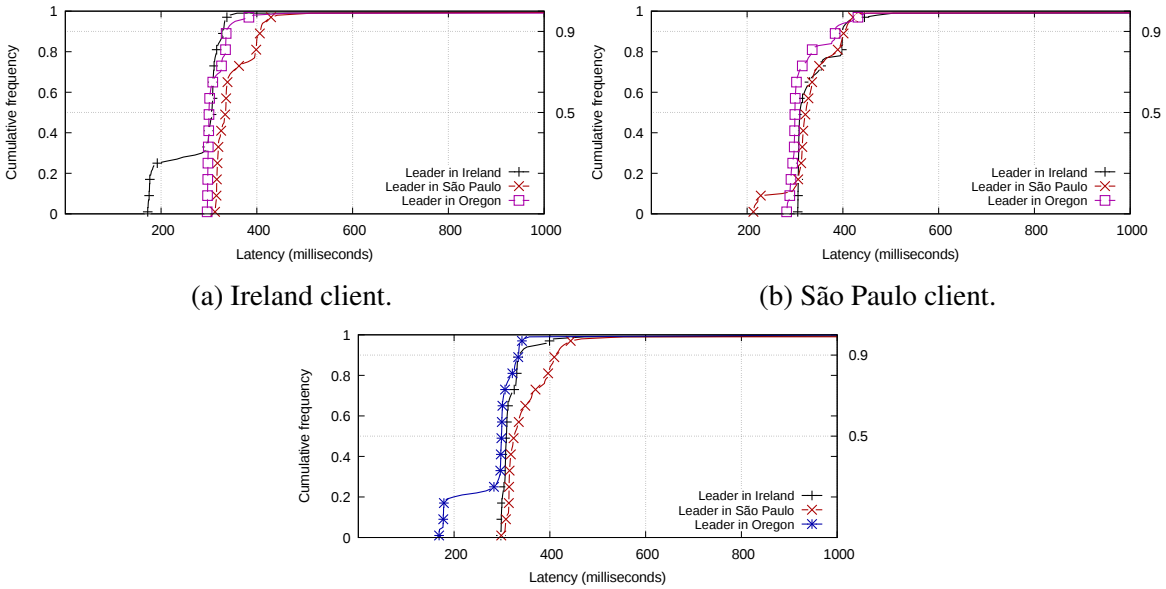
(a) Ireland client.

(b) São Paulo client.

(c) Oregon client.

(d) Sydney client.

Figure 5.7: Cumulative frequency distribution of latencies observed by each client when the leader is placed across Amazon EC2 regions (BFT mode).



(a) Ireland client.

(b) São Paulo client.

(c) Oregon client.

Figure 5.8: Cumulative frequency distribution of latencies observed by each client when the leader is placed across Amazon EC2 regions (CFT mode).

Location	BFT mode				CFT mode		
Client \ Leader	Wroclaw	Madrid	Munich	London	Wroclaw	Madrid	Munich
Wroclaw	**152/178**	149/186	118/268	160/290	**65/72**	66/119	66/104
Madrid	153/203	**148/160**	119/297	145/266	66/100	**66/84**	66/161
Munich	153/188	149/177	**113/249**	145/276	66/100	77/113	**66/72**
London	151/200	148/200	132/185	**111/248**	-	-	-
Overall	153/191	149/177	111/230	146/270	66/100	67/111	66/131
Improvement	0%/7%	0%/10%	-2%/-8%	24%/8%	2%/28%	1%/24%	0%/45%

Table 5.5: Client latencies' 50th/90th percentile (milliseconds) when the leader is placed across PlanetLab hosts.

Location	BFT mode				CFT mode		
Client \ Leader	Ireland	São Paulo	Oregon	Sydney	Ireland	São Paulo	Oregon
Ireland	**609/655**	634/710	587/618	636/679	**306/330**	335/407	300/336
São Paulo	610/658	**634/708**	589/618	637/686	310/400	**322/404**	300/384
Oregon	610/657	634/712	**588/617**	637/682	309/336	325/412	**299/334**
Sydney	612/656	633/717	587/612	**636/680**	-	-	-
Overall	610/656	634/712	588/603	636/682	308/359	327/408	300/339
Improvement	0%/0%	0%/0%	0%/-2%	0%/0%	0%/8%	15%/1%	0%/1%

Table 5.6: Client latencies' 50th/90th percentile (milliseconds) when the leader is placed across Amazon EC2 regions.

Table 5.6 shows the median and 90th percentile observed in each Amazon EC2 region. The comparison of the latency observed by the clients co-located with the leader (in bold) with the overall latency observed by all clients shows a benefit of at most 8% (CFT mode with the leader in Ireland) in having co-location, regardless of the region. However, having the leader in Oregon results in a slightly lower 90th percentile for all clients.

Main conclusion: Since the obtained results depict a similar trend in the two different testbeds, we can assert that co-locating clients with the leader does not necessarily improve the latency of replicated state machines. On the other hand, placing the leader in the host with better connectivity with the remaining replicas can yield more consistent improvements. More precisely, the benefit of reaching the leader faster is not as important as hosting the leader in the replica with faster links with others.

5.2.6 Discussion

The results presented in Section 5.2.2 indicate that, as expected, bypassing communication steps reduces client latency in BFT SMR protocols. However, even though read-only (resp. speculative) executions are up to 63% (resp. 35%) faster than standard executions, the benefits of tentative and fast executions are not so impressive: about 20% and 10%, respectively. The difference, as explained before, is due to the fact that fast executions requires larger quorums than tentative execution, which requires waiting for more messages (that can be slow in an heterogeneous environment such as a WAN). In the end, tentative execution matches the theoretically expected benefits: by avoiding 20% of the communication steps (see Figure 5.1), we did obtain a latency reduction of approximately 20%.

The results of Sections 5.2.3 and 5.2.4 show that decreasing the ratio between the number of expected messages and the total number of replicas can decrease latency significantly, especially for CFT replication. More specifically, clients that wait less replies had a 90th percentile latency improvement of up to 36% (resp. 11%) in CFT (resp. BFT) mode; and adding more replicas to the system while maintaining the same quorum size brings improvements of up to 72% (resp. 17%) in CFT (resp. BFT) mode. These results are mainly due to the performance-heterogeneity of hosts and links in real wide area networks: if the latency between all replicas were similar and network delivery variance were small, the observed improvements would be much more modest. Furthermore, they are in accordance with other studies showing that using smaller quorums may bring better latency than decreasing the number of communication steps (e.g., Junqueira *et al.* (2007)).

The results of Section 5.2.5 indicates that having the leader close to a client will not significantly reduce the SMR latency for this client. This result is unexpected since several protocols implement mechanisms such as rotating coordinator (Mao *et al.*, 2008; Veronese *et al.*, 2010) and multiple proposers (Moraru *et al.*, 2013) to make each client submit its requests to the closest replica. We found two main explanations for this apparent contradiction. First, the heterogeneity of real environments such as PlanetLab and Amazon EC2 make optimizations for reducing latency less effective. In fact, the authors of Mencius acknowledge that the protocol achieves lower latency than Paxos only in networks with small latency variances (Mao *et al.*, 2008). Second, in CFT mode, BFT-SMART clients wait for replies from a majority of replicas to ensure linearizability due to the use of the read-only optimization. EPaxos, Mencius and Paxos clients wait only for a single reply from the leader.

This means that client-leader co-location in these protocols potentially reduce the latency in two communication steps, while in BFT-SMaRt this reduction is in only one (clients still need to wait for at least one additional reply). Consequently, having a client co-located with the leader should decrease the number of communication steps by 25% in CFT mode and 20% in BFT mode, while in Mencius and EPaxos such theoretical improvement can reach 50%. Moreover, its worth to point out that these benefits appear only in favorable conditions. For example, EPaxos presents almost the same latency of Paxos when under high request interference (Moraru *et al.*, 2013).

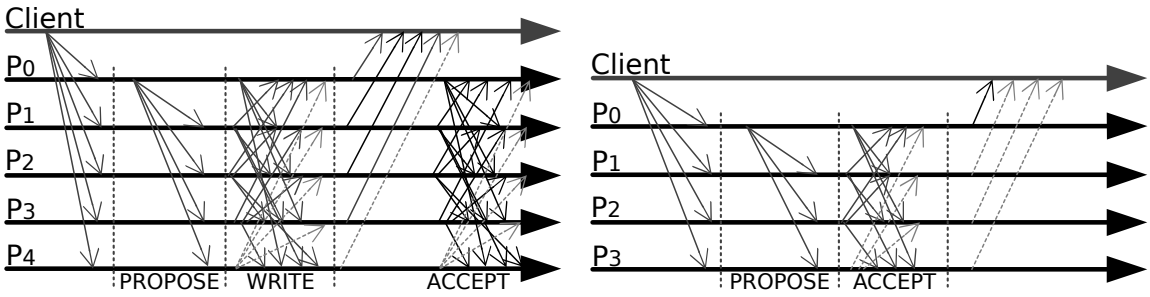
As a final remark, it is worth noting that our results show that having a leader in a well-connected replica brings, in general, more benefits than having clients co-located with leaders. For instance, we observed that latency was usually lower when the leader replica was hosted in Madrid, rather than when the leader replica was placed in the same location as a particular client. In the same line, adding faster replicas to the system may significantly improve latency, as shown in Section 5.2.4. For example, the addition of Madrid to the set of replicas decreased the 90th percentile latency in Oslo and Aveiro by 39% and 72%, respectively (CFT mode). More generally, these results highlight the fact that *not all replicas are the same in geo-replication* and that both the leader location and quorum formation rules must take into account the characteristics of the sites being used.

5.3 The WHEAT Protocol

This section describes WHEAT, a WAN-optimized SMR protocol implemented on top of BFT-SMaRt. We start by discussing the WAN optimizations employed in our protocol and then we introduce two novel vote assignment schemes for using smaller quorums without endangering the safety of SMR. We conclude the section with an evaluation of WHEAT in Amazon EC2.

5.3.1 Deriving the protocol

WHEAT employs the optimizations that were most effective in improving the latency of SMR in WANs. The selected optimizations (discussed below) reduce the number of communication steps, the number of replies that clients wait, and the ratio between the quorum size and the total number of replicas. Since the results of client-leader co-location were not



(a) BFT (b) CFT

Figure 5.9: WHEAT's message pattern for $f = 1$ and one additional replica.

so expressive, and given that its implementation would require substantial changes in the base SMR protocol (which is already complex enough, as discussed in previous chapters), we rejected this optimization and followed the fixed leader approach. As with BFT-SMART, WHEAT can be used in BFT or CFT modes, implementing the message patterns illustrated in Figure 5.9.

Reducing the number of communication steps: In BFT mode, WHEAT employs the read-only and tentative execution optimizations introduced in PBFT (Castro & Liskov, 2002). The reason to support tentative executions instead of fast or speculative executions is as follows: (1) during our experiments, tentative executions displayed slightly better latency than fast executions (i.e., they had a lower 90th percentile); (2) speculative executions are useful in environments were the network is predictable and stable, which we cannot expect in many geo-distributed settings. If such conditions are not met by the network (i.e, not delivering replies from all replicas within the required time window), clients need to trigger the commit phase and force the protocol to execute five communications steps (Kotla *et al.*, 2009); and (3) tentative executions do not require modifications to the synchronization phase of BFT-SMART. Fast executions would required modifications to account for cases where a value was decided solely with $\lceil \frac{n+3f+1}{2} \rceil$ *WRITE* messages, whereas the rollback operation can be triggered using the state transfer protocol already implemented in BFT-SMART. Furthermore, usage of speculative executions would demand the complete re-implementation of the original protocol, to account for the several corner cases necessary to preserve correctness under this type of executions, such as the aforementioned commit phase. Another advantage of tentative executions is that *ACCEPT* messages can be piggybacked in the next *PROPOSE* or *WRITE* messages, similarly to PBFT (Castro & Liskov, 2002).

Reducing the number of replies a client waits: In BFT mode, the use of read-only and tentative executions lead WHEAT clients to always gather responses from a Byzantine quorum of replicas, i.e., at least $\lceil \frac{n+f+1}{2} \rceil$ replies. This means that it is impossible to enforce the optimization evaluated in Section 5.2.3 without giving up linearizability (Herlihy & Wing, 1990). However, single-reply read-only executions can still be used in the CFT mode as long as clients always contact the leader replica.[1] Consequently, in CFT mode WHEAT clients only need to wait for one reply (from any replica during write operations and from the leader during read-only operations).

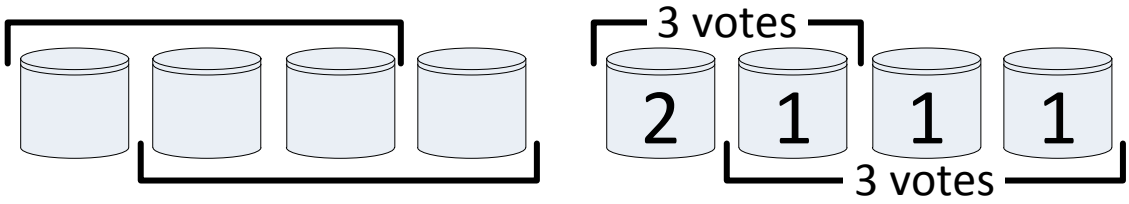
Reducing the ratio between the quorum size and the number of replicas: As observed in Section 5.2.4, it is possible to significantly decrease latency by adding more replicas to the system, as long as the quorums used in the protocol remain with the same size. Both the Byzantine and crash variants of WHEAT are designed to exploit this phenomenon by modifying the quorum requirements of the protocol. However, to avoid violating the safety properties of traditional SMR protocols, we need to introduce a mechanism to secure the formation of intersecting quorums of variable size. In the next section we introduce a voting scheme that preserves this requirement.

5.3.2 Vote assignment scheme

Our voting assignment schemes integrate the classical ideas of weighted replication (Garcia-Molina & Barbara, 1985; Gifford, 1979; Pâris, 1986) to SMR protocols. The goal is to extend quorum-based SMR protocols to (1) rely primarily on the fastest replicas present in the system; and (2) preserve its original safety and liveness properties.

The most important guarantees that quorum-based protocols need to preserve are (1) all possible quorums overlap in some correct replica and (2) even with up to f failed replicas, there is always some quorum available in the system. In CFT protocols like Paxos (Lamport, 1998), quorums must overlap in at least one replica. Such intersection is enforced by accessing a simple majority of replicas during each communication step of a protocol. More specifically, protocols access $\lceil \frac{n+1}{2} \rceil$ replicas out of $n \geq 2f + 1$. BFT protocols like PBFT (Castro & Liskov, 2002), on the other hand, usually employ disseminating Byzantine quorums (Malkhi & Reiter, 1998) with at least $f + 1$ replicas in the intersection. In this case, protocols access $\lceil \frac{n+f+1}{2} \rceil$ replicas out of $n \geq 3f + 1$. With this strategy, adding a single extra

[1] It is also necessary to use leases on the client, since the leader can be demoted at any point.



(a) Classical (always a majority of replicas). (b) Weighted (from $f + 1$ to $n - f$ replicas).

Figure 5.10: Quorum formation when $f = 1$ and $n = 4$ (CFT mode).

replica to the system results in higher latency, since any possible quorum becomes larger in size – unlike the weighted quorums strategy we present below.

The fundamental observation that we make is that accessing a majority of replicas guarantees the aforementioned intersection, but that this is not the only way to secure such intersection. More specifically, if n is greater than $2f + 1$ (in CFT mode), it is possible to distribute weights across replicas in such way that a majority is not always required to (correctly) make progress. As an example, consider the quorums illustrated in Figure 5.10 (with one extra replica in the system). Whereas in Figure 5.10a the intersection is obtained by strictly accessing a majority of replicas, in Figure 5.10b we see that we can still obtain an intersection with a variable number of replicas (since we can obtain a sum of 3 votes by either accessing 2 or 3 replicas). In particular, if the replica with weight 2 is successfully probed, the protocol can finish a communication step with a quorum comprised by only half of the replicas. Otherwise, a quorum comprised by all replicas with weight 1 is necessary to make progress. Notice that for this distribution to be effective, it is necessary to attribute weight 2 to the fastest replica in the system.

We now generalize the weight distribution proposed in Figure 5.10b to account for other values of f. The objective is to assign certain numbers of votes (i.e., weights) to each replica in accordance with their connectivity/performance. This vote assignment must be done carefully to ensure that minimal quorums composed by faster replicas will be used under normal conditions (i.e., when the faster replicas are indeed faster) and larger, yet available quorums can be used to ensure that up to f faulty replicas are tolerated (despite their weights).

Let Q_v be the minimum number of votes that a quorum of replicas must hold to guarantee that quorums overlap by at least one correct replica. A quorum is said to be *safe and minimal* (or just minimal) if it is comprised by only $f + 1$ replicas that together hold Q_v votes.

This quorum size is minimal because if f or less replicas were considered a quorum, other intersecting quorums would require *more than* $n - f$ replicas. These quorums will not be available when there are f faulty replicas in the system. This means that having quorums with less than $f + 1$ replicas implies giving up consistency or availability, as described in classical quorum definitions (Malkhi & Reiter, 1998). In a BFT system, for the same reasons, a minimal quorum must be comprised of $2f + 1$ replicas.

Using the above definitions, we consider vote distribution schemes that satisfy the following properties:

- **Safe minimality:** There exists at least one minimal quorum in the system.
- **Availability:** There is always a quorum available in the system that holds Q_v votes.
- **Consistency:** All quorums that hold Q_v votes intersect by at least one correct replica.

In the following, we deduce vote assignment schemes for CFT and BFT modes that abide by these properties. The complete correctness proofs (both for CFT and BFT) can be found in Appendix C.

CFT vote distribution: To calculate the vote distribution under CFT mode, we start by introducing the parameter Δ, which represents the number of extra replicas available in the system. Thus, n can be calculated using Δ as follows:

$$n = 2f + 1 + \Delta \tag{5.1}$$

We now introduce two additional variables. N_v represents the sum of the number of votes $\sum V_i$ that are attributed to each replica i. F_v is the maximum number of votes that can be dismissed in the system. Having these parameters, we can apply the standard quorum rules to the votes instead of the replicas. Hence, N_v is calculated as follows:

$$N_v = \sum V_i = 2F_v + 1 \tag{5.2}$$

As an example, consider Figure 5.10b: the sum of all votes adds up to 5, which represents an *abstract quorum system* comprised by 5 hosts capable of withstanding 2 faults. Therefore, for this case, $N_v = 5$ and $F_v = 2$.

Since Δ and f are our input parameters, we need to (1) find a relation between Δ and f and values for N_v, F_v and V_i; (2) use those variables to force the emergence of *replica*

quorums that intersect by one replica. More precisely, votes must be distributed in such a way that once $F_v + 1$ votes are gathered, a quorum of replicas always overlap with other quorums by at least one correct replica. Therefore, the value of Q_v is calculated as follows:

$$Q_v = F_v + 1 \tag{5.3}$$

If we assume that only two possible values can be assigned to replicas (e.g., a binary vote distribution), as in Figure 5.10b, we can introduce variables V_{max} and V_{min}. However, we need to find how many replicas are assigned V_{max} and V_{min}. Let u be the number of replicas holding V_{max} votes and, consequently, $n - u$ the number of replicas holding V_{min}. Since the sum of all votes must be equal to N_v, we have:

$$N_v = 2F_v + 1 = uV_{max} + (n - u)V_{min} \tag{5.4}$$

In the example of Figure 5.10b, $V_{max} = 2$, $V_{min} = 1$ and $Q_v = F_v + 1 = 3$. We can observe two cases where 3 votes can be obtained: either by (1) accessing the single V_{max} replica and one of the V_{min} replicas, or (2) accessing all V_{min} replicas. Notice that in both cases, the same number of votes is dismissed, but not the same number of replicas; in case (1) two replicas are ignored, but in case (2) only one replica is left unprobed. Also note that the number of votes dismissed is 2, which happens to be the value of F_v (as we pointed out previously). This indicates that F_v has a direct relation to V_{max} and V_{min}. Given this observation, we generalize this example scenario to represent any Δ and f:

$$F_v = (\Delta + f)V_{min} = fV_{max} \tag{5.5}$$

We can derive the relation between V_{max} and V_{min} from (5.5) as follows:

$$V_{max} = \frac{(\Delta + f)}{f} V_{min} \tag{5.6}$$

If we assume $V_{min} = 1$, equations (5.5) and (5.6) become:

$$F_v = \Delta + f \tag{5.7}$$

$$V_{max} = \frac{\Delta + f}{f} = 1 + \frac{\Delta}{f} \tag{5.8}$$

Having now more refined formulas for F_v, V_{max} and V_{min}, we can return to equation (5.4) and obtain the value of u:

$$2(\Delta + f) + 1 = u(1 + \frac{\Delta}{f}) + (n - u) \Rightarrow u = f \tag{5.9}$$

Knowing that $u = f$, still by equation (5.4), there must be f replicas holding V_{max} votes and $n - f$ replicas holding 1 vote (since $V_{min} = 1$). We thus have our CFT vote assignment scheme: equations (5.7) and (5.8) give us the values for F_v and V_{max} respectively, all in function of Δ and f.

The main benefit of this scheme is that if all the f replicas holding V_{max} are probed faster than any other, then just one of the $\Delta + f + 1$ other replicas holding V_{min} votes will be disregarded (like the two-replica quorum of Figure 5.10b). However, in the worst case, if f replicas holding V_{max} votes fail (or are slow), then all replicas with V_{min} votes will be accessed instead (as the three-replica quorum of Figure 5.10b). Since we obtained this scheme making 3 main assumptions (abstract quorums, binary distribution and $V_{min} = 1$) we give out a complete proof of correctness in Appendix C.

BFT assignment: The reasoning here is similar to the CFT scheme, but with the following differences. First, equations (5.1) and (5.2) become $n = 3f + 1 + \Delta$ and $N_v = \sum V_i = 3F_v + 1$, respectively. These equations still lead to the same values of F_v and V_{max}, but u becomes $2f$ instead of f. This forces the system to have $2f$ replicas holding V_{max} and $\Delta + f + 1$ replicas holding one vote (V_{min}). Moreover, it is necessary to gather $2F_v + 1$ votes on each quorum, which makes $Q_v = 2F_v + 1$. Finally, a minimal quorum must be comprised by $2f + 1$ replicas instead of $f + 1$. Like for the CFT mode, a complete proof of correctness is available in Appendix C.

Improving latency: These weight assignment schemes can improve the latency of a system by allowing more choice: if there is a spare replica in the system that is (or becomes) faster than the rest, the optimal quorum (e.g., 2 out-of 4 replicas as in Figure 5.10b) would contain this replica. In normal protocols this benefit would be smaller because quorums always increase. For instance, Paxos *always* requires $\lceil \frac{n+1}{2} \rceil$ messages to make progress; but if our voting scheme were introduced to Paxos, it could make progress collecting *as less as* $f + 1$ messages. In the case of Byzantine protocols such as PBFT, the least number of messages required to make progress would be $2f + 1$ if our scheme were implemented. It is worth mentioning that in the event that the systems experiences a period of high load, it is possible that the minimal quorum becomes overloaded and unable to reply faster than other quorums, thus forcing the system to make progress with different quorums. Nonetheless, any SMR protocol based on quorum systems is subject to this issue.

Dynamically assigning weights: Our voting assignment schemes dictate that the subset of the faster replicas hold V_{max} votes. However, such subset can change at runtime as the network conditions change. If suddenly any of the fastest replicas become slow or unavailable, the votes can be re-distributed so that other replicas take the place of the ones that are no longer the fastest. This can be done using BFT-SMART's reconfiguration protocol (described in Section 4.2.3.3) to re-distribute the votes and even change the location of the leader for improving the performance of the system. Our current implementation assumes an external monitoring and administration entity is used to change the vote assignment, but we posit this can be done at runtime if some kind of self-monitoring is employed (which we left for future work).

Increased fault tolerance with spare replicas: SMR protocols such as PBFT (Castro & Liskov, 2002) or Paxos (Lamport, 1998) stop making progress if more than f replicas are unavailable. On the other hand, the additional spare replicas used in WHEAT can be used to allow the system to continue executing even if more than f hosts are crashed or slow. However, this is possible only for certain subsets of unavailable replicas. This happens because, using this voting scheme, each phase of the protocol completes after gathering a majority of votes instead of probing a majority of replicas. For instance, if in the example shown in Figure 5.10b, two replicas V_{min} fail, the protocol would continue to work, since the remaining V_{max} and V_{min} replicas would still satisfy $F_v + 1$. Generally speaking, if more than f replicas holding a total of no more than F_v votes fail, the protocol will still execute correctly.[1]

Besides that, in case a fault is reliably detected, the same SMR reconfiguration protocol used to change the replica group or (re)assign votes to replicas can be used by an administrator to remove the failed replica(s) from the system and reassign votes to preserve the desired fault tolerance threshold f. In the end, up to $f + \Delta$ faulty replicas can be tolerated if such reconfigurations are performed in a timely way, i.e., ensuring that there is never more than f faulty replicas simultaneously in the system.

Notice that our approach is better than using BFT-SMART's reconfiguration protocol to replace unavailable replicas. Such replacement would require a state transfer, which can be a slow operation for large state sizes and limited wide-area links. For example, a 4GB-state will take more than fifty minutes to be transferred in a 10 Mbps network (better than several links between EC2 regions). With our approach, the extra replicas are already active and up-to-date in the system, so the reconfiguration takes approximately the time to execute a

[1] In the case of BFT mode, this only holds true if all those replicas fail strictly by crash.

Sites	Ireland	São Paulo	Oregon	Sydney	Virginia
Ireland	0	211 ± 10	171 ± 11	340 ± 11	88 ± 10
São Paulo	208 ± 14	0	217 ± 19	359 ± 4	123 ± 3
Oregon	171 ± 14	217 ± 11	0	205 ± 7	70 ± 12
Sydney	336 ± 26	359 ± 4	205 ± 10	0	255 ± 12
Virginia	88 ± 10	123 ± 4	71 ± 13	256 ± 5	0

Table 5.7: Average *roundtrip* latency and standard deviation (milliseconds) between Amazon EC2 regions as measured during a 24 hour-period.

"normal" SMR operation.

Finally, our assignment schemes could also be used to assign a higher number of votes to replicas on more reliable and available sites (instead of the faster ones), improving thus the reliability and availability of the system in lieu of latency.

5.3.3 Implementation and Evaluation

We implemented WHEAT by extending BFT-SMART for supporting the chosen optimizations (Section 5.3.1) and considering replicas with different number of votes (Section 5.3.2). This required around two hundred additional lines of (Java) code, with most of the modifications related with a new module that takes into account the weights of replicas for calculating the quorums used in the protocol.

We evaluated WHEAT by running a set of experiments in Amazon EC2 and comparing the results with the original BFT-SMART system. As in the EC2 experiments reported in Section 5.2.5, we use sites on Ireland, Oregon, Sydney and São Paulo (only in BFT mode) for BFT-SMART using also Virginia as the additional replica of WHEAT. This means that the original version of BFT-SMART employed 4 replicas in BFT mode (resp. 3 in CFT mode) whereas WHEAT employed 5 replicas in BFT mode (resp. 4 in CFT mode), with two of these replicas in North America. In the BFT mode, the following parameters were employed (obtained through the voting schemes described previously): $N_v = 7$, $F_v = 2$, $V_{max} = 2$ for the replicas in Oregon and Virginia. In the CFT mode, the configuration was $N_v = 5$, $F_v = 2$, $V_{max} = 2$ for the replica in Virginia. We attributed the V_{max} values to the these sites because they were the ones with better connectivity to others, as shown in Table 5.7.

Figure 5.11 illustrates the latencies' cumulative frequency distribution across all clients. The median and 90th percentile latencies for each client location and protocol is presented in Table 5.8.

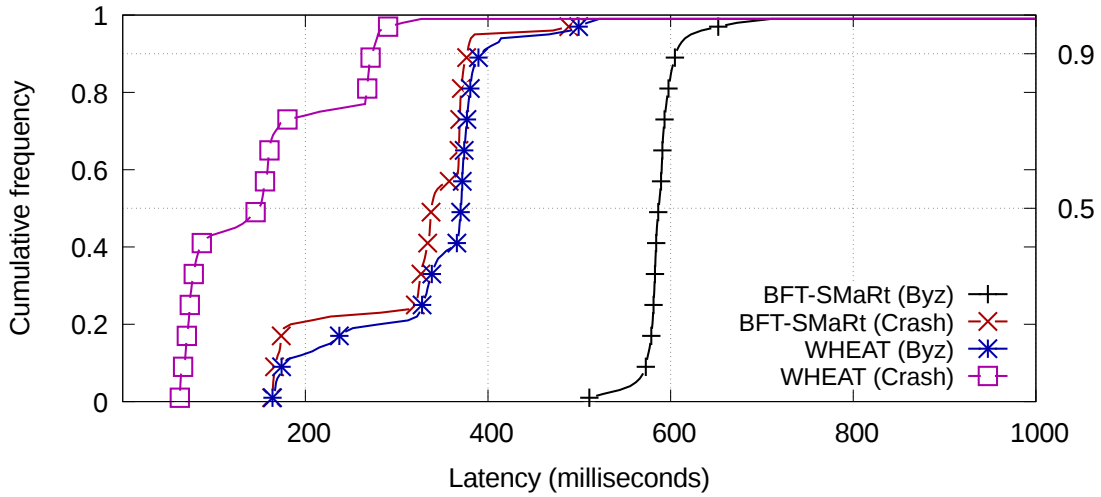


Figure 5.11: Cumulative frequency distribution of latencies for WHEAT and BFT-SMART in Amazon EC2 with the *leader in Oregon*.

Client Region	BFT mode		CFT mode	
	BFT-SMaRt	WHEAT	BFT-SMaRt	WHEAT
Oregon (L)	**587/605**	**178/336**	**174/336**	**72/92**
Ireland	587/609	368/390	338/373	160/191
Sydney	586/600	375/470	370/386	270/297
São Paulo	587/609	377/482	-	-
Virginia	-	357/383	-	76/142
Overall	587/606	370/393	338/377	149/273
Improvement	-	**37%/35%**	-	**56%/28%**

Table 5.8: 50th/90th percentile latencies (milliseconds) observed by BFT-SMART and WHEAT clients in different regions of Amazon EC2 with the *leader in Oregon*.

By employing the selected optimizations (Section 5.3.1) and using an additional replica in Virginia without increasing the quorum requirements (i.e., three and two replicas for BFT and CFT, respectively), WHEAT achieves a 90th percentile latency improvement of 35% (BFT) and 28% (CFT), when compared with BFT-SMART. The overall median latency improved even more with WHEAT: 37% in BFT and 56% in CFT. Interestingly, the client in the leader region (Oregon) observed improvements even more impressive, with median latency values matching the roundtrip times between Oregon and Ireland (BFT mode) or

Virginia (CFT mode). This is a consequence of the fact that this client is co-located with the leader *in the most well-connected site of the system*.

The improvements shown in these experiments for WHEAT should be taken with a bit of salt since they may be due the use of an additional site with a good roundtrip latency with other replicas (see Table 5.7). If the new replica used in WHEAT were added on an hypothetical Amazon EC2 region "moon" (instead of Virginia), with a higher roundtrip latency with all other sites, the WHEAT results would be less impressive since the faster quorums will be the same of BFT-SMaRt. The only benefits will be due to the other optimizations (tentative executions for BFT and single-reply for CFT) implemented in the system. Nonetheless, our results illustrate the fact that in a real geo-replication setup there are significant benefits in assigning different values to different replicas. Furthermore, even with the required algorithmic support, it is important to choose the location of the spare replicas employed in WHEAT, to ensure the smaller quorums will bring significant benefits.

A note on throughput: WHEAT aims to improve geo-replication latency, and thus all of its optimizations target this performance metric. However, the fact it uses Δ more replicas than BFT-SMaRt, implies it might achieve a slight lower peak throughput than the original system. This happens because more replicas lead to more message transmissions, which results in higher CPU and network bandwidth utilization. More precisely, each consensus instance on BFT-SMaRt requires the exchange of $3f + 18f^2$ (resp. $2f + 4f^2$) messages in BFT mode (resp. CFT mode), whereas in WHEAT it requires $3f + \Delta + 2(3f + \Delta)^2$ (resp. $2f + \Delta + (2f + \Delta)^2$) message exchanges. Although undesirable, this drawback will only affect a saturated system, which is rarely the case in production environments. Moreover, as discussed in Section 5.2.1, throughput can be improved by increasing CPU and network resources, while latency can only be addressed by better protocols.

5.4 Additional Related work

Weighted replication: This approach was originally proposed by Gifford (1979), and then revisited by Garcia-Molina & Barbara (1985) and Pâris (1986). While Gifford made all hosts hold a copy of the state with distinct voting weights, Pâris made a distinction between hosts that hold a copy of the state and hosts that do not hold such copy, but still participate in the voting process (thus acting solely as witnesses). More recent works have confirmed the usefulness of these ideas also for performance by showing that adding few servers to a group of replicas can significantly improve the access latency of majority quorums (Bakr &

Keidar, 2008), and the same kind of technique is being used in practical systems to improve tolerance to slow servers (Dean & Barroso, 2013). By contrast, Garcia-Molina & Barbara (1985) address the idea of weighted replication for *coterie systems*, which later evolved into the classic quorum systems without including vote distribution. Unlike our approach, none of these works target geo-replication: Pâris (1986) and Garcia-Molina & Barbara (1985) are strictly theoretical contributions and Gifford (1979) considers a local datacenter. In addition, as far as we are aware of, we present the first vote assignment scheme that unpacks a weight distribution in function of the expected number of faults and the amount of spare replicas available in the system, as well as demonstrating the correctness of the scheme.

WAN measurements: Quorum systems are building blocks for the most practical SMR protocols (e.g., Castro & Liskov (2002); Lamport (1998); Mao *et al.* (2008); Moraru *et al.* (2013); Veronese *et al.* (2010)), which makes it important to predict their availability and performance in wide-area environments.

Amir & Wool (1996) proposed the earliest empirical evaluation of quorum systems that we are aware of. Their approach consisted of gathering uptime data from a real system deployed in multiple hosts. These hosts were scattered across two distinct sites which communicated with each other over the internet. The results suggest that machine crashes are correlated, network partitions are frequent, and a whole-system crash is a rare, yet possible event.

Whereas Amir & Wool investigated the *availability* of a quorum system, Bakr & Keidar (2002) investigate the *latency* of distributed algorithms over the internet. Unlike Amir & Wool, the hosts were geographically distributed across more than two sites. The authors observed the message loss rate over the internet was not negligible and protocols with high message complexity displayed higher loss rate. However, those experiments did not evaluated quorum systems, i.e., each communication round was considered finished only when every host of the system replied to every message.

The authors conducted further research in (Bakr & Keidar, 2008) considering quorum systems and how the number of hosts probed by a client impacts latency and availability. Their results suggest that majority quorums do not perform well with small probe sets. Moreover, they also claim that increasing the probe size by as few as a single host can reduce latency by a considerable margin. Our results confirm these claims.

There are also studies which compare the performance of different total order broadcast protocols - a fundamental building block for SMR - over a WAN (e.g., Anker *et al.* (2003); Cason *et al.* (2015); Ekwall & Schiper (2007); Schiper *et al.* (2009)). The exper-

iments present in this chapter have a different goal: instead of evaluating the performance of distinct protocols, we compare geo-replication-related optimizations employed by different protocols, but implemented in the same codebase, to validate the effectiveness of these optimizations in real WANs.

5.5 Concluding Remarks

In this chapter we revisited some optimizations proposed in the literature for improving the latency of SMR protocols in wide-area networks. More concretely, we implemented such optimizations in a modified version of BFT-SMART and compared its latency with a non-modified version in the PlanetLab testbed and Amazon EC2 cloud to assess which of these optimizations bring significant benefits. Our results indicated that removing communications steps and demanding less replies from replicas lead to latency reductions of up to 20%, depending on the hosts and fault model. Surprisingly, using the closer replica as the leader held less benefits than what was expected.

These results guided our design for WHEAT, an SMR protocol optimized for geo-replication that can be configured either for crash-only or Byzantine fault tolerance. WHEAT was implemented by extending BFT-SMART with the optimizations we observed as most effective and implementing a novel vote assignment strategy for efficient quorum usage. The evaluation of WHEAT in Amazon EC2 showed gains of up to 56% for certain configurations, when compared with the unmodified BFT-SMART. In the next chapter, we further evaluate WHEAT's and BFT-SMART's performance within the context of two practical systems.

6 Applications

In this chapter we explore the impact that BFT-SMaRt and WHEAT can have in practical systems. In particular, we consider the following applications: a middleware that augments the resilience of typical transactional databases and an open-source blockchain platform. We describe how BFT-SMaRt/WHEAT was integrated into these systems and present experimental evaluations for both of them on a local and geo-distributed networks.

6.1 Transactional Databases

Our first use case is a BFT database replication middleware that implements Byzantium (Garcia *et al.*, 2011), a transactional protocol which allows transactions to execute concurrently, but eschews the need for a trusted node. This protocol is attractive for three reasons: (1) it is capable of providing the robustness of Byzantine fault tolerance within acceptable loss of performance; (2) it allows usage of *off-the-shelf* database management systems (DBMS) without the need to modify their inner functionality, as long as the DBMS' supports transactional processing and *snapshot isolation semantics* (Berenson *et al.*, 1995); and (3) it uses a BFT total order broadcast primitive as a black box, thus rendering it ideal as a stress-test for BFT-SMaRt and WHEAT.

Byzantium's architecture is depicted in Figure 6.1. An application issues operations to replicas using a client shim. The shim implements most of Byzantium's protocol logic and uses a proxy to provide total ordering of COMMIT and ROLLBACK operations. This proxy is also used to disseminate all other operations to the replicas using best-effort broadcast.

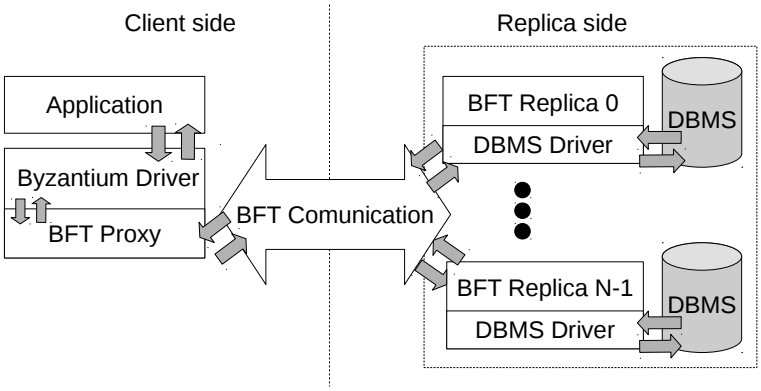


Figure 6.1: Byzantium's architecture.

Each replica hosts an instance of the chosen DBMS. Servers maintains a complete copy of the database on their DBMS' instance. Communication between the BFT Replica and the server's DBMS is done via a driver compatible to DBMS being used.

6.1.1 The Byzantium Protocol

Byzantium assumes a system model similar to BFT-SMaRt's and most typical BFT state machine protocols: it requires $3f + 1$ replicas to withstand up to f faults, as well as an unbounded, yet finite amount of malicious clients, all executing in a partially synchronous environment. During normal case executions (i.e., in the absence of faults), clients and replicas interact as shown in Figure 6.2. The most important feature of the protocol is the fact that only two types of operations need to be propagated using total order. More precisely, each client starts (resp. finishes) a transaction by issuing a BEGIN (resp. COMMIT/ROLLBACK) operation via total order broadcast. All other operations invoked within a transaction are transmitted to all replicas using best-effort broadcast.[1] However, only one of them can execute operations as soon as they arrive and reply to the client. This replica is chosen and agreed upon system start-up and is designated as the *master replica*.[2] Non-master replicas only execute operations at end of the transaction. Moreover, upon transmission of a COMMIT/ROLLBACK, clients include the order by which all the operations were executed at the

[1]However, all operations from a given client must enforce FIFO order. This is needed because, otherwise, replicas might receive operations for a given transaction before receiving the BEGIN operation for that same transaction.

[2]Byzantium's master replica does not need to be the same as the leader process of the BFT total order protocol.

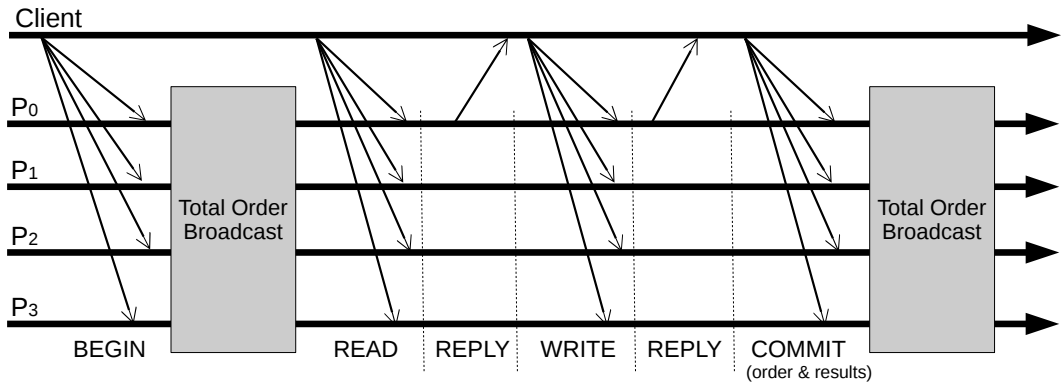


Figure 6.2: The Byzantium protocol.

master, as well as the cryptographic hashes of the results received from those operations. During the COMMIT, all other replicas execute the operations using the order supplied by the client, compute the cryptographic hashes of the results generated from their own execution, and compare them with the hashes received from the client. If a replica determines that it received all operations associated with the transaction and that the hashes of their own results match the ones given by the client, they locally commit the operations. Otherwise, the transaction is rolled back and the client is notified about the misbehavior.

Byzantine behavior displayed by a non-master replica can be masked in the same way as in BFT-SMART. On the other hand, the master could try to disrupt the system by issuing erroneous results to clients and/or refusing to reply altogether. Nonetheless, the system can withstand such erroneous replies because all correct replicas compare their own operation results with the ones provided by the client. If the master fails to reply, clients will suspect it and ask for a master change using the total order broadcast. In order to prevent Byzantine clients from triggering divergent database states at different replicas, the total order primitive needs to be augmented with the mechanism for masking non-determinism proposed by Castro *et al.* (2003), re-purposed for allowing replicas to determine whether or not they hold a sequence of operations matching the sequence provide by any given client. If $2f + 1$ replicas agree (resp. disagree) about holding such sequence, then the total order primitive will deliver a COMMIT (resp. ROLLBACK) operation. If there is no set of $2f + 1$ replicas either agreeing or disagreeing about a given sequence, the leader of the total order broadcast is changed and the process is repeated.

One thing that is worth stressing about Byzantium is the fact that its normal case execution only requires total ordering among BEGIN and COMMIT/ROLLBACK operations.

The reason why this is possible is because the system demands the DBMS to provide snapshot isolation (SI) semantics (Berenson *et al.*, 1995). This is a level of isolation that forces each transaction to provide clients with a snapshot of the database's state obtained at the moment the transaction began, i.e., when the BEGIN operation was executed. Each update performed on a transaction is visible only within that transaction, and are only applied to the database after the COMMIT operation. In case two concurrent transactions issue conflicting updates, only one is allowed to commit, while the other is forced to abort. Since all BEGIN and COMMIT/ROLLBACK operations are totally ordered, all correct replicas see the same sequence of snapshots. Because of this, it is safe to execute the operations only at the master and delay the execution at non-masters until commit time.

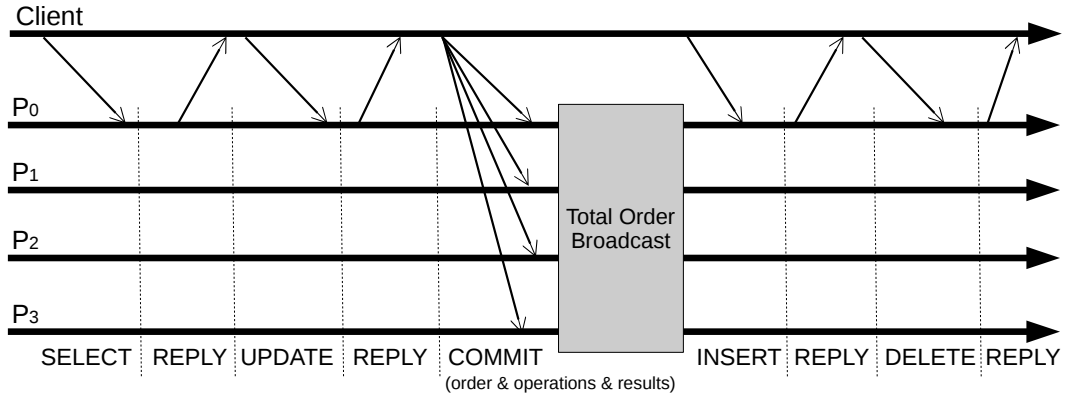
Since most of the communication between clients and replicas is done using best-effort broadcast and clients only wait for a single reply from the master, read and update latency is the same as in typical non-replicated systems. The performance penalty is therefore encapsulated in the BEGIN and COMMIT/ROLLBACK operations, since it is were the total order broadcast is invoked. However, this is done not only for efficiency, but because SI may lead concurrent transactions to experience conflicting updates. Without employing total order across all operations, such conflicting updates may be executed in different order across the different replica, thus causing transactions to abort non-deterministically.

6.1.2 Implementation

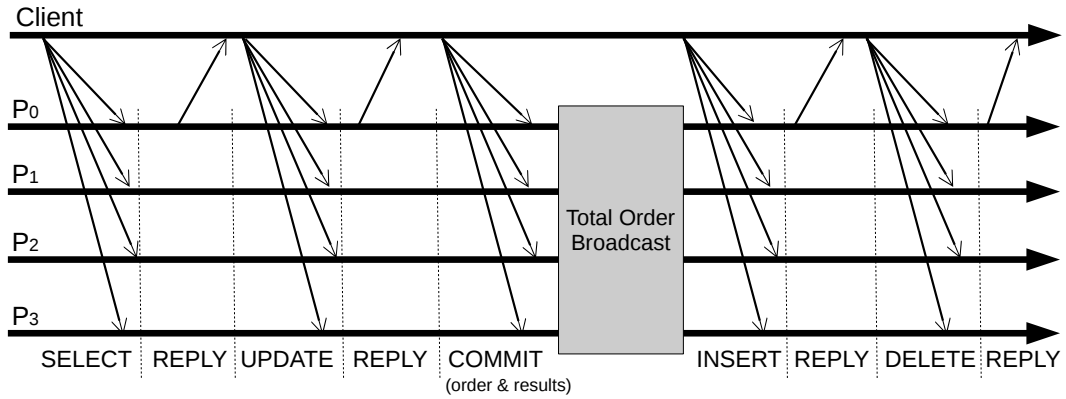
Byzantium is implemented in an open-source Java codebase called SteelDB (Santos, 2014). This codebase is fully compliant with the JDBC specification and runs a simplified variant of the protocol described in Section 6.1.1, which is illustrated in Figure 6.3a. The two main simplifications are the following: (1) there is no need for a BEGIN operation, as the JDBC specification mandates that new transactions are immediately started after a commit or rollback of a previous transaction from the same client; and (2) clients only send their operations to the master instead of broadcasting them to all replicas. The other replicas only receive the operations when clients issue COMMIT or ROLLBACK operations.

For this thesis, SteelDB's codebase was refurbished to (1) use the most recent version of BFT-SMART's Java library; (2) support PostgreSQL [1] as its DBMS; and (3) withstand heavier workloads than what it was originally designed to expect. In addition, we also developed a variant of the codebase that implements the protocol illustrated in Figure 6.3b. This version

[1] https://www.postgresql.org/



(a) Normal.



(b) Optimized.

Figure 6.3: SteelDB protocol.

is more similar to Byzantium's standard protocol due to the fact that it not only broadcasts operations to all replicas, but also implements the optimization which allows for non-masters to speculatively execute operations before COMMIT time. To correctly and safely allow for concurrent transactions to speculatively execute conflicting updates across all replicas, these transactions must be forced to obtain the same set of locks on all replicas. Enforcing this without access to the internals of the DBMS requires operations to be received in the same order on all replicas. To achieve this without using total order broadcast, each correct non-master must employ a mechanism similar to the commit barriers proposed by Vandiver *et al.* (2007). This mechanism dictates that an operation op_n from transaction t_m can execute at non-masters under the following conditions: (1) it receives operation op_{n+1} from t_m, and (2) Transaction t_{m-1} was commited.

6.1.3 Evaluation

The goal of this evaluation is to measure SteelDB's performance both in a local area network and in a geo-replicated environment when using BFT-SMART and WHEAT as BFT total order broadcast primitive.

6.1.3.1 Setup

Local area experiments were deployed within up to 6 virtual machines within a single Amazon EC2 region, while the geo-replicated scenario was comprised by the same regions used in Section 5.3.3: Oregon, Ireland, Sydney and São Paulo (four BFT-SMART replicas), with Virginia standing as WHEAT's additional replica (five replicas). Weight distribution is also the same (Oregon and Virginia weighted V_{max}, all others weighted V_{min}). However, because these experiments are more computing-intensive, we used instances of the type *t2.medium* rather than *t2.nano*.

The setup was similar across both scenarios: SteelDB was set to withstand $f = 1$ (thus requiring a minimum of 4 replicas), using PostgreSQL 9.6 and Ubuntu 14.04. We evaluated both normal and optimized versions of the codebase, using BFT-SMART and WHEAT. We also conducted experiments with a standard deployment of PostgreSQL 9.6. In the case of the geo-replicated scenario, we placed the clients in an additional virtual machine running in the same region as the master (hosted in Oregon).

Measurements were obtained using an open-source benchmark based on the TPC-C specification.[1] TPC-C is an on-line transaction processing (OLTP) benchmark which simulates a wholesale parts supplier that operates out of a number of warehouses and their associated sales districts (TPC, 2010). This benchmark issues 3 types of read-write transactions designated *NEW-ORDER* (the main transaction of TPC-C), *PAYMENT* and *DELIVERY*, as well as 2 types of read-only transactions designated *ORDER-STATUS* and *STOCK-LEVEL*. We set TPC-C to use a database comprised of 50 warehouses and launched 30 terminals using the benchmark's default workload (92% updates and 8% reads). All experiments executed for 5 minutes and were repeated 3 times.

As discussed in Section 6.1.1 the performance penalty associated with Byzantium is reflected on the time it takes to complete a COMMIT operation. Therefore, besides gathering TPC-C's results about client throughput, we have adapted the benchmark to also collect

[1] https://sourceforge.net/projects/benchmarksql/

data about the commit latency of *NEW-ORDER* transactions, as well measurements for the duration of the entire transactions.

6.1.3.2 Results

Local area. Figure 6.4 presents the throughput observed by TPC-C's terminals.[1] As expected, standard PostgreSQL outperforms all SteelDB configurations, with normal SteelDB with standard BFT-SMART being the least performant.

Optimized SteelDB with standard BFT-SMART improved performance significantly and made the system reach 50% of the throughput of standard PostgreSQL. This is because this codebase allows non-masters to speculatively execute operations, which results in less operations to be executed at commit time. Optimized SteelDB with WHEAT also displayed better performance, but the gain was negligible. This is to be expected, since WHEAT's weight distribution scheme is designed for wide-area environments comprised of heterogeneous links. This is not the case on a local datacenter, which is ought to be a homogeneous environment. One could expect to still observe some kind of improvement due to WHEAT's tentative execution, but because the latency of the *WRITE* and *ACCEPT* communication step is negligible in a local network (in the order of microseconds when compared to the milliseconds observe in wide-area), the improvement is negligible.

Figure 6.5 presents the latency of *NEW-ORDER* operations as observed by TPC-C's terminals. The observed latencies are consistent with the throughput results discussed previously and support the claim that Byzantium's loss of performance is directly dependent from the commit latency. Across both percentiles, commit latency is reduced by more than 75% when using the optimized codebase, which results in a decrease in transaction latency.

Geo-replication. Figure 6.6 presents TPC-C's observed throughput when the replicas are spread around the world. Under this scenario, WHEAT's impact on performance is much more significant, increasing throughput by more than 35% in relation to the optimized SteelDB with BFT-SMART. This is because WHEAT's weight distribution scheme is designed to be effective in a heterogeneous environment such as a wide-area scenario. In addition, WHEAT's tentative executions also display a larger impact because the *WRITE*/*ACCEPT* phases of the protocol have a larger overhead in wide-area. Latency results for *NEW-ORDER*

[1] *tpmC* stands for the throughtput associated with *NEW-ORDER* operations, while *tpmTOTAL* represents the throughput from all types of operations combined.

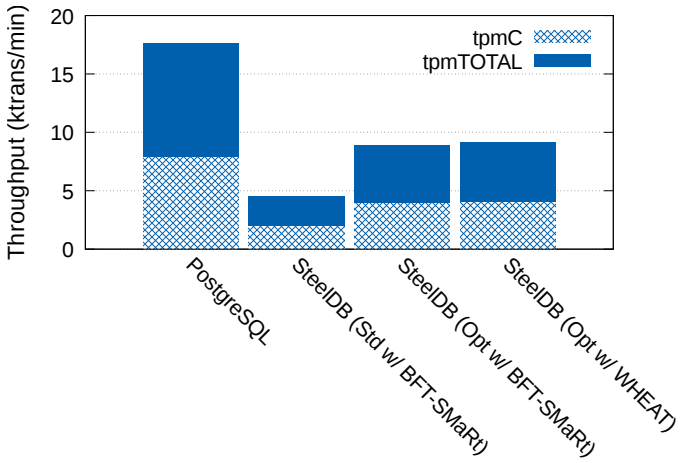
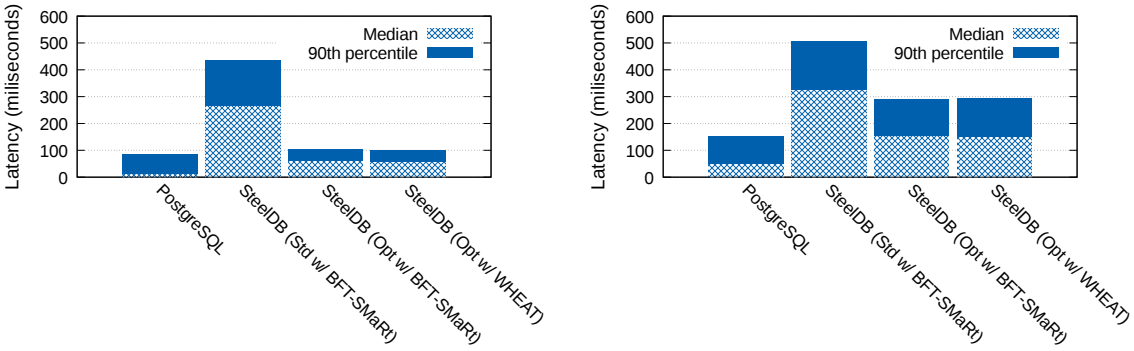


Figure 6.4: SteelDB throughput (local area).



(a) Commit.

(b) Transaction.

Figure 6.5: *NEW-ORDER* latency (local area).

operations are shown in Figure 6.7. These measurements are also consistent with the aforementioned throughput results: median latency (resp. 90th percentile) is reduced by more than 35% (resp. 20%) when using the optimized codebase.

6.1.4 Discussion

As expected, state machine replication adds a large overhead to the performance of a replicated database when compared to a non-replicated setup. Nonetheless, this overhead can be significantly mitigated by employing optimizations to the transactional protocol, and in the

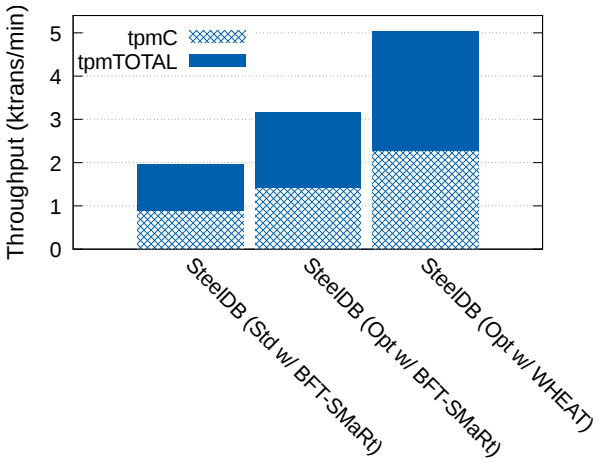
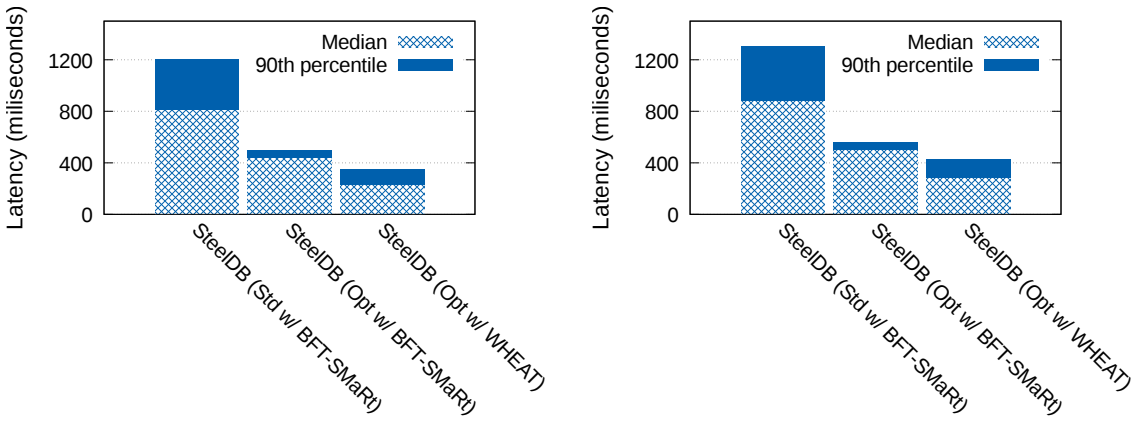


Figure 6.6: SteelDB throughput (Geo-distributed).



(a) Commit.

(b) Transaction.

Figure 6.7: *NEW-ORDER* latency (Geo-distributed).

case of wide-area, by deploying a total order broadcast designed to operate in such scenario. More precisely, the implemented optimizations resulted in a combined performance gain of 50% in local-area and 60% in wide-area when compared to the non-optimized counterparts. As a result, *NEW-ORDER* latency decreased to less than 500 milliseconds on both scenarios, which still falls within acceptable limits for user interaction with production systems (Card *et al.*, 1991; Miller, 1968; Nielsen, 1993).

6.2 Permissioned Blockchains

Our second application scenario considers the integration of BFT-SMART and WHEAT into a platform for permissioned blockchains named Hyperledger Fabric.[1] We start by describing the fundamentals of blockchain technology (Section 6.2.1) and proceed to present the platform (Section 6.2.2). After that, we describe the BFT-SMART/WHEAT ordering service (Section 6.2.3) and present an experimental evaluation of the system (Section 6.2.4).

6.2.1 Blockchain Technology

A blockchain is an open database that maintains a distributed ledger typically deployed within a peer-to-peer network. It is comprised by a continuously growing list of records called *blocks* that contain transactions (Nakamoto, 2009). Blocks are protected from tampering by cryptographic hashes and a consensus mechanism.

The structure of a blockchain – illustrated in Figure 6.8 – consists of a sequence of blocks in which each one contains the cryptographic hash of the previous block in the chain. This introduces the property that block j cannot be forged without also forging all subsequent blocks $j+1...i$. Furthermore, the consensus mechanism is used to (1) prevent the whole chain from being modified; and to (2) decide which block is to be appended to the ledger.

The blockchain may abide by either the *permissionless* or *permissioned* models (Vukolić, 2015). Permissionless ledgers are maintained across peer-to-peer networks in a totally decentralized and anonymous manner (Nakamoto, 2009; Wood, 2015). In order to determine which block to append to the ledger next, peers need to execute a Proof-of-Work (PoW) consensus (Garay *et al.*, 2015). The key idea behind PoW consensus is to limit the rate of new blocks by solving a cryptographic puzzle, i.e., execute a CPU intensive computation that takes time to solve, but can be verified quickly. This is achieved by forcing peers to find a nonce N such that given their block B and a limit L, the cryptographic hash of $B||N$ is lower than L (Back, 2002; Dwork & Naor, 1993). The first peer that presents such solution gets its block appended to the ledger. Roughly speaking, as long as the adversary controls less than half of the total computing power present in the network, PoW consensus prevents the adversary from creating new blocks faster than honest participants.

Permissionless blockchains have the benefit of enabling the ledger to be managed in a completely open way, i.e., any peer willing to hold a copy of the ledger can try to create new

[1] https://www.hyperledger.org/

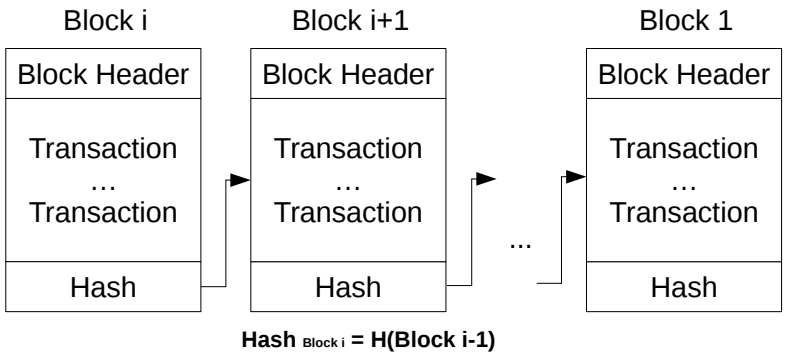


Figure 6.8: Blockchain structure.

blocks for it. On the other hand, the computational effort associated to PoW consensus is both energy- and time-consuming; even if specialized hardware is used to find a Proof-of-Work, this mechanism still imposes a limit on transaction latency.

By contrast, permissioned blockchains employ a closed consortium of nodes tasked with creating new blocks and executing a traditional Byzantine consensus protocol to decide the order by which the blocks are inserted to the ledger (Buchman, 2016; Cachin & Vukolic, 2017a; Martino, 2016). Hence, permissioned blockchains do not expend the amount of resources that open blockchains do and are able to reach better transaction latency and throughput. In addition, it makes possible to control the set of participants tasked with maintaining the ledger – rendering this type of blockchain a more attractive solution for larger corporations, since it can be separated from the dark web or illegal activities.

6.2.2 Hyperledger Fabric

Hyperledger Fabric (or simply, Fabric) (Androulaki *et al.*, 2018) is an open-source project within the Hyperledger collaborative effort.[1] It is a modular permissioned blockchain system designed to support pluggable implementations of different components, such as the ordering and membership services. Fabric enables clients to manage transactions by using chaincodes, endorsing peers and an ordering service.

Chaincode is Fabric's counterpart for smart contracts (Szabo, 1996). It consists of code deployed on the Fabric's network, where it is executed and validated by the endorsing peers, who maintain the ledger, the state of a database (modeled as a versioned key/value store), and abide by endorsement policies. The ordering service is responsible for creating blocks

[1] https://www.hyperledger.org/

for the distributed ledger, as well as the order by which each blocks is appended to the ledger.

6.2.2.1 Fabric protocol

The Fabric general transaction processing protocol (Androulaki *et al.*, 2018) – depicted in Figure 6.9 – works as follows:

1. *Clients create a transaction and send it to endorsing peers*. This message is a signed request to invoke a chaincode function. It must include the chaincode ID, timestamp and the transaction's payload.

2. *Endorsing peers simulate transactions and produce an endorsement signature*. They must verify if the client is properly authorized to perform the transaction by evaluating access control policies of a chaincode. Transactions are then executed against the current state. Peers transmit to the client the result of this execution (read and write sets associated to their current state) alongside the endorsing peer's signature. No updates are made to the ledger at this point.

3. *Clients collect and assemble endorsements into a transaction.* The client verifies the endorsing peers signatures, determine if the responses have the matching read/write set and checks if the endorsement policies has been fulfilled. If these conditions are met, the client creates a signed *envelope* with the peers' read and write sets, signatures and the Channel ID. A channel is a private blockchain on a Fabric network, providing data partition. Each peers of the channel share a channel-specific ledger. The aforementioned envelope represents a *transaction proposal*.

4. *Clients broadcast the transaction proposal to the ordering service.* The ordering service does not read the contents of the envelope; it only gathers envelopes from all channels in the network, orders them using atomic broadcast, and creates signed chain blocks containing these envelopes.

5. *The blocks of envelopes are delivered to the peers on the channel.* The envelopes within the block are again validated to (1) ensure the endorsement policies were fulfilled, and (2) to check if there were changes to the peers' state for read set variables (since the read set was generated by the transaction execution). To this end, the read set contains a set of versioned keys that endorsing peers read at the time of simulating

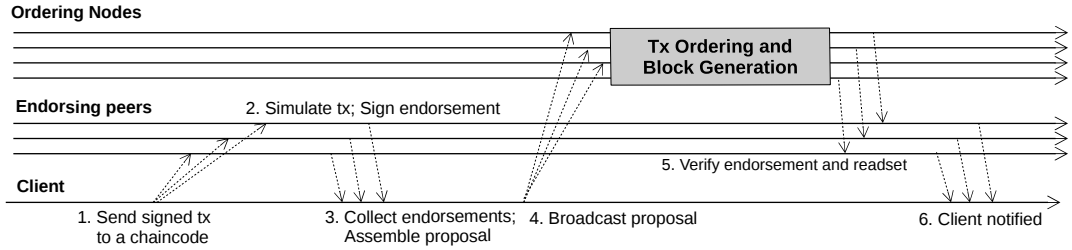


Figure 6.9: Hyperledger Fabric transaction processing protocol (Androulaki *et al.*, 2018).

a transaction (step 2). Depending on the success of these validations, the transaction proposal contained in envelopes are marked as either being valid or invalid.

6. *Peers append the received block to the channel's blockchain.* For each valid transaction, the write sets are committed to the peers' current state. An event is triggered to notify the client that the transaction has been immutably appended to the channel's blockchain, as well as notification of whether the transaction were deemed valid or invalid. Notice that invalid transactions are also added to the ledger, but they are not executed at the peers. This also has the added benefit of making it possible to identify malicious clients, since their actions are also recorded.

An important aspect of the Fabric protocol is that endorsement (step 2) and validation (step 5) can be done at different peers. Furthermore, contrary to the chaincode execution during endorsement, the validation code needs to be deterministic, i.e., the same transaction validated by different peers in the same state produces the same output (Androulaki *et al.*, 2018).

6.2.2.2 Pluggable Consensus

As mentioned before, Fabric is a modular blockchain system. In particular, one of the components that support plug-and-play capability is the ordering service. Currently, Fabric's codebase includes the following ordering service modules: (1) a centralized, non-replicated ordering service that does not execute any distributed protocol that is used mostly for testing the system; and (2) a replicated ordering service capable of withstanding crash faults, consisting of an Apache Kafka cluster[1] and its respective ZooKeeper ensemble (Hunt *et al.*, 2010). At the time of this writing, both modules have limitations. The non-replicated module

[1] https://kafka.apache.org/

requires very few hardware resources, but it is also a single point of failure. The Kafka-based module is both decentralized and robust, but can only withstand crash faults.

6.2.3 BFT-SMaRt Ordering Service

The BFT-SMaRt module for Fabric's ordering service consists of an ordering cluster and a set of frontends. The ordering cluster is composed by a set of $3f+1$ nodes that collect envelopes from the frontends and execute the BFT-SMART's replication protocol with the purpose of totally ordering these envelopes among them. Once a node gathers a predetermined number of envelopes, it creates a new block containing these envelopes and a hash of the previously created block, generates a digital signature for the block, and disseminates it to all known frontends, which collect $2f + 1$ matching blocks from ordering nodes. The $2f + 1$ blocks are necessary because frontends do not verify signatures. However, this number guarantees a minimum of $f+1$ valid signatures to peers and clients.[1] Frontends are part of the peer trust domain and are responsible for (1) relaying the envelope to the ordering cluster on behalf of the client, and (2) receiving the blocks generated by the ordering cluster and relaying them to the peers responsible for maintaining the distributed ledger.

6.2.3.1 Architecture

BFT-SMART's ordering service architecture is illustrated in Figure 6.10. The frontend is composed by the Fabric codebase and a BFT shim. The Fabric codebase (implemented in Go) provides an interface for Fabric clients to submit envelopes. These envelopes are relayed to the BFT shim using UNIX sockets. This shim is implemented in Java and maintains (1) a client thread pool that receive envelopes and relays them to the ordering cluster, and (2) a receiver thread that collects blocks from the cluster. Envelopes (resp. blocks) are sent to (resp. received from) the cluster through the BFT-SMaRt proxy. The proxy does that by issuing an asynchronous invocation request to the BFT-SMART client-side library, ensuring it does not block waiting for replies. To ensure that the shim performs computations on equivalent data structures to the Fabric codebase, the ordering service uses the Hyperledger Fabric Java SDK to parse and assemble data structures used in Fabric.

[1] If the frontends are programmed to perform signature verification, only $f + 1$ matching blocks suffice.

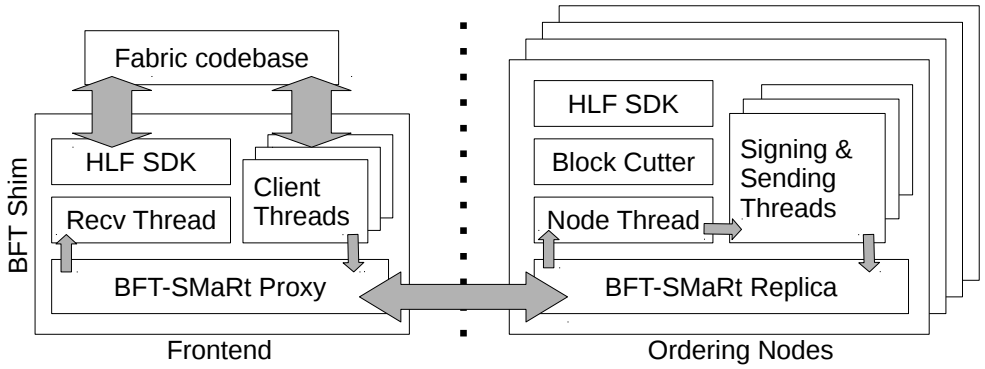


Figure 6.10: BFT-SMaRt ordering service architecture.

6.2.3.2 Batching

The ordering nodes are implemented on top of the BFT-SMART service replica, thus receiving a stream of totally ordered envelopes. Each node maintains an object named *blockcutter*, where the envelopes received from the service replica are stored before being assembled into a block. The blockcutter is responsible for managing the envelopes associated to each Fabric channel and creating a batch of envelopes to be included in a block for the ledger associated to that channel. We implement this batching mechanism instead of relying on BFT-SMART's native batching because (1) each BFT-SMART's batch may contain envelopes that are not associated to the same channel, which means the envelopes cannot be all assembled into the same block; (2) Fabric supports *configuration envelopes*, which are supposed to remain isolated from regular envelopes; and (3) Fabric's native batching policies are not equivalent to BFT-SMART's (for instance, Fabric imposes a batching limit based on its size in terms of bytes, whereas BFT-SMART limit is based on number of requests per batch). Once the blockcutter holds a pre-determined number of envelopes for a channel (the block size), it notifies the node thread that it is time to drain its envelopes and create the next block.

6.2.3.3 Parallelization

After the blockcutter is drained, a sequence number is assigned to the future block and submitted to the signing/sending thread pool alongside with the respective block header. This header contains the aforementioned sequence number and the cryptographic hashes from the previous header and the hash for the block's envelopes. Notice that this thread pool does not cause non-determinism across the nodes because (1) the block header and envelopes to be assigned to new blocks are generated sequentially within the node thread, and (2) the

only structures that each node needs to maintain as part of the application state is the block header from the previous iteration of the node thread. Similarly to the frontend, the Fabric Java SDK is used to correctly handle and create the data structures used by the system. In addition, this SDK is also used to generate cryptographic hashes and ECDSA (Elliptic Curve DSA) signatures (Johnson & Menezes, 1998) that can be validated by other components of Fabric. Once the block is created and signed, it is transmitted to all active frontends. This is done through a custom *replier* (supported by the extensible API of BFT-SMART) that, instead of sending the operation result (i.e., the generated block) to the invoking client, sends it to a set of registered BFT-SMART clients (i.e., the frontends).

6.2.3.4 Application State

The state maintained by the ordering nodes is comprised by the headers for the last block associated to each channel, information about the current configuration of channels, and the envelopes currently stored at the blockcutter. Since the headers have a constant size and the envelopes are periodically drained from the blockcutter, the state maintained at the ordering nodes will always be bounded and remain smaller than the size of the ledger maintained by Fabric peers.

6.2.3.5 Validation and Reconfiguration

One last aspect of this service relates to channel reconfiguration and transaction validation. Fabric's architecture is resilient to blocks contained junk transactions, hence ordering services can avoid performing transaction validation. In the particular case of our ordering service, transactions can be validated by the signing/sending threads prior to generating block signatures. Transactions can then be removed from the block if the validation fails. The exception to this is a special category of transactions that are used to perform channel reconfiguration. These transactions need to be validated and executed prior to submitting them to a blockcutter.

6.2.4 Evaluation

In this section we describe the experiments conducted to evaluate BFT-SMART's ordering service and discuss the observed results. Our aim here is not to evaluate the whole Fabric system, but only the ordering service, which may typically be the bottleneck of the system.

6.2.5 Parameters affecting the Ordering Performance

The throughput of the ordering service (i.e., the rate at which envelopes are added to the blockchain TP_{os}) is bounded by one of three factors: a) the rate at which envelopes are ordered by BFT-SMART ($TP_{bftsmart}$) for a given envelope size, number of envelopes per block and number of receivers; b) the number of blocks signed per second (TP_{sign}); or c) the size of the generated blocks. These parameters are illustrated in Figure 6.11.

Given an envelope size es, block sizes bs, and a number of receivers r (i.e., the frontends to which the ordering nodes transmit the generated blocks), the peak throughput of the ordering service is bounded as follows:

$$TP_{os}^{bs,es,r} \leq min(TP_{sign} \times bs, TP_{bftsmart}^{bs,es,r}) \tag{6.1}$$

An important remark is that this equation considers that a block is signed only once by each ordering node. However, in Fabric a blocks need to be signed twice. The second signature is needed to attach the block transaction to an execution context (details are out of the scope of this chapter). If this is the case for the considered application, the TP_{sign} term used in the equation must be replaced by $\frac{TP_{sign}}{2}$.

6.2.6 Signature Generation

In order to estimate TP_{sign}, we executed a very simple signature benchmark program written in Java in one of the machines used in Chapter 4: a Dell PowerEdge R410 server, which possesses two quad-core 2.27 GHz Intel Xeon E5520 processor with hyper-threading (thus having 16 hardware threads) and 32 GB of memory. The server runs Ubuntu 14.04 with JVM 1.8.0. Our program spawns a number of threads to create ECDSA signatures for blocks of fixed size and calculates how many of such signatures are generated per second.

Results. Our results show that our server can generate up to 8.4k signatures/sec, when running with 16 threads (Figure 6.12). Furthermore, the effect of the block size is mostly negligible as the ECDSA signature is computed over the hash of the block. These results, together with the fact that a blocks are expected to contain 10+ envelopes in Fabric, lead us to conclude that signature generation is not expected to be a bottleneck in our setup.[1]

[1]For example, by using blocks with $bs = 100$ envelopes, we can sign up to $TP_{sign} \times bs = 840k$ envelopes/sec.

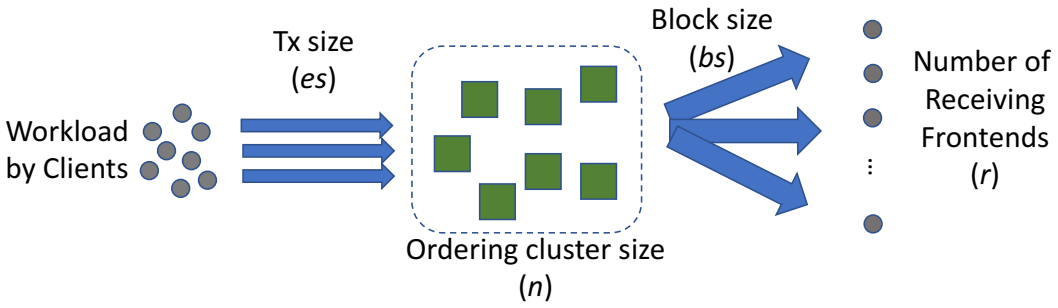


Figure 6.11: Ordering service performance model.

6.2.7 Ordering Cluster in a LAN

The experiments aims to evaluate the BFT-SMART ordering service by using clients that emulate the behavior of multiple ordering service frontends. They were executed with clusters of 4, 7, and 10 nodes, withstanding 1, 2, and 3 Byzantine faults, respectively. Furthermore, we also fiddled with the block size, by configuring each cluster configuration to assemble blocks containing either 10 or 100 envelopes (i.e., transactions). This is meant to observe the behaviour of each cluster when throughput is bound by either the rate of signature generation or by the rate of envelope reception. The environment is the same used in Chapter 4: Dell PowerEdge R410 servers connected through a Gigabit ethernet.

For each micro-benchmark configured to have x nodes and y envelopes/block, we gathered results for (1) envelopes with different sizes, and (2) a variable number of receivers. More precisely, each envelope size is representative of submitting to the ordering cluster: (1) a SHA-256 hash (40 bytes); (2) three ECDSA endorsement signatures (200 bytes); and (3) transaction messages of 1 and 4 kbytes. Considering the way Fabric 1.0 operates, the values related with (3) are more representative of the size of a transaction. In particular, our limited experience shows that transactions compressed with gzip tend to be usually close to 1 kbyte. Nonetheless, measurements for (1) and (2) are important to show the potential of the ordering service if different design choices were taken in future versions of Fabric.

Measurements for the throughput associated to block generation were gathered at ordering node 0 (the leader replica of BFT-SMART's replication protocol). To reach the system's peak throughput, each execution was performed using 16 to 32 clients distributed across 2 additional machines. We also repeated the micro-benchmark with 4 nodes with blocks of 100 envelopes. All experiments used 16 signing threads (to match the number of available cores) and were repeated 3 times taking 5 minutes each.

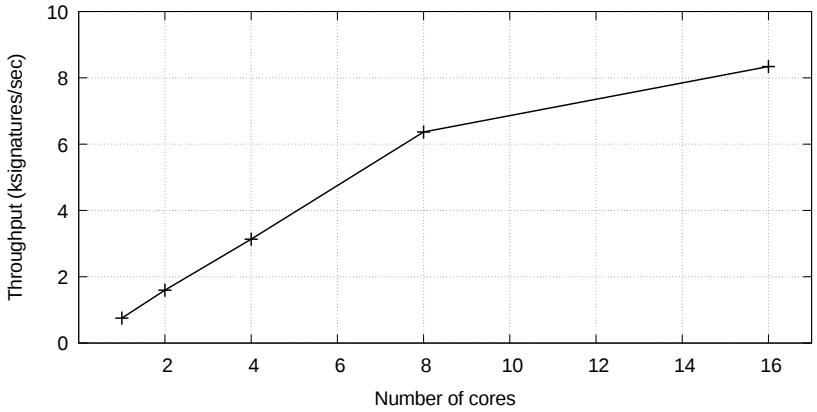
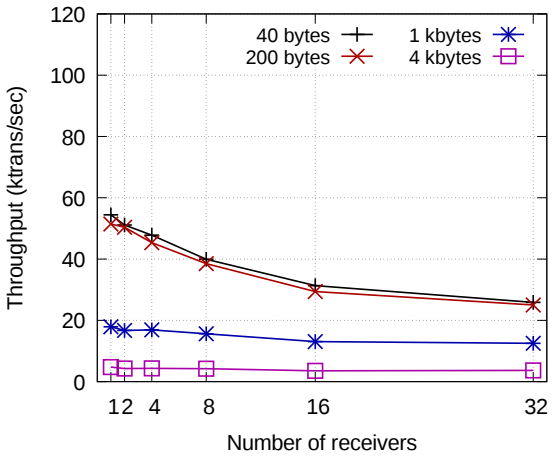


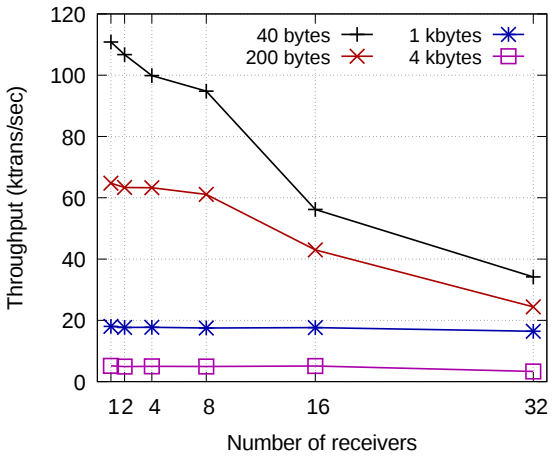
Figure 6.12: Signature Generation for Fabric blocks.

Results. The obtained results for local-area are presented in Figure 6.13. Even though throughput drops when increasing the number of receivers, the impact of the number of receivers is considerably smaller for larger transactions (1k and 4 kbytes). This is because for these envelope sizes, the overhead of the replication protocol is greater than the overhead of transmitting blocks of 10 and 40 kbytes. In particular, since the batch limit of the BFT-SMaRt is set to 400 requests (default value), the PROPOSE message of the underlying replication protocol can have up to 0.4-1.6MBs with these envelope sizes.

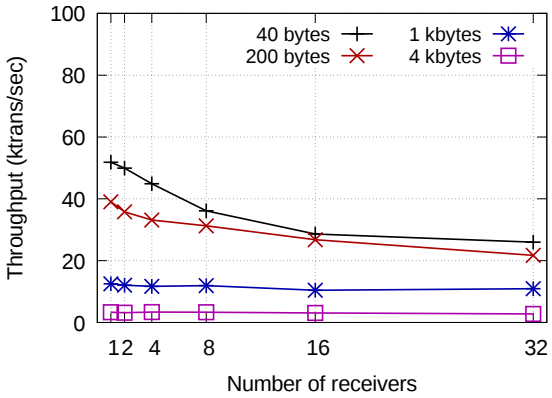
It can be observed that when using 10 envelopes/block (Figures 6.13a, 6.13c, and 6.13e), the maximum throughput observed is approximately 50k transactions/second (when there exists only 1 to 2 receivers in the system), which is way below the $8.4k \times 10 = 84k$ envelopes/sec capacity if only signatures are considered (Section 6.2.6). This can be explained by the fact that signature generation needs to share CPU power with the replication protocol, hence creating a thug-of-war between the application's worker threads and BFT-SMaRt's I/O threads and queues – in particular, BFT-SMaRt alone can take up to 60% of CPU usage when executing a void service with asynchronous clients. Hence, the performance drops when compared to the micro-benchmark from Section 6.2.6, which was executed in a single machine, stripped of the overhead associated with BFT-SMaRt. Moreover, for up to 2 receivers and envelope sizes of 1 and 4 kbytes, the peak throughput becomes similar to the results observed in Chapter 4. This is because for these request sizes BFT-SMaRt is unable to order envelopes at a rate equal to the rate at which the system is able to produce signatures.



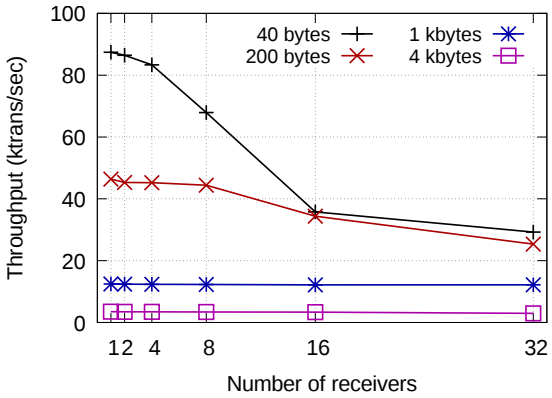
(a) 4 orderers, 10 envelopes/block.



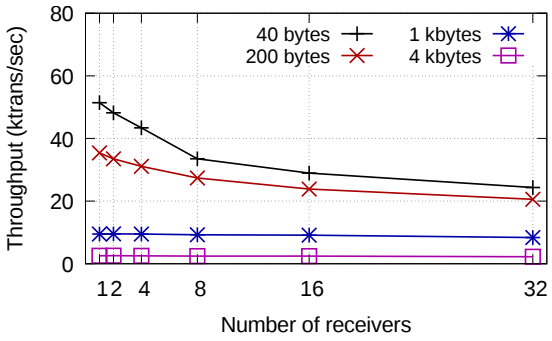
(b) 4 orderers, 100 envelopes/block.



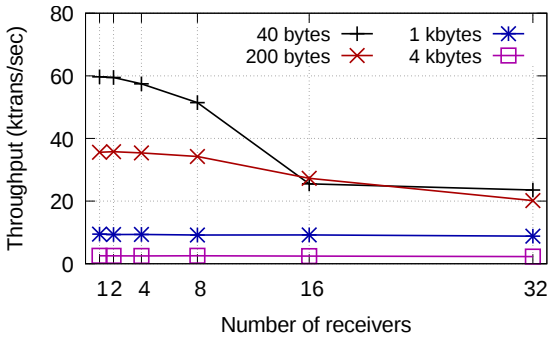
(c) 7 orderers, 10 envelopes/block.



(d) 7 orderers, 100 envelopes/block.



(e) 10 orderers, 10 envelopes/block.



(f) 10 orderers, 100 envelopes/block.

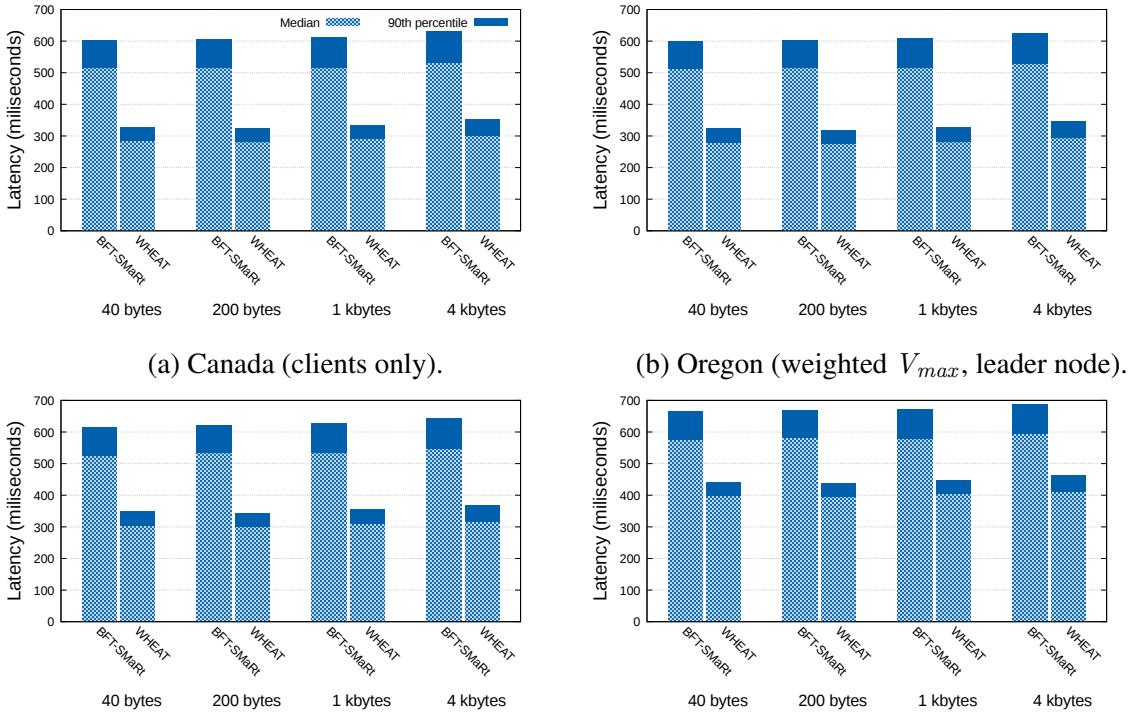
Figure 6.13: BFT-SMART Ordering Service throughput for different envelope, block and cluster sizes.

Figures 6.13b, 6.13d, and 6.13f show the results obtained for 100 envelopes/block, when each node is not subject to CPU exhaustion. It can be observed that, across all cluster sizes, throughput is significantly higher for smaller envelope sizes and up to 8 receivers. This happens because even though each node creates blocks at a lower rate – approximately 1100 blocks per seconds – each block contains 100 envelopes instead of only 10. Moreover, this configuration makes the rate at which envelopes are ordered to become similar to the rate at which blocks are created. This means that for smaller envelope sizes, it is better to adjust the nodes' configuration to avoid consuming all the CPU time and rely on the rate of envelope arrival. However, for envelopes of 1 and 4 kbytes the behavior is similar to using 10 envelopes/block, specially from 7 nodes onward. This is because for larger envelope sizes – as discussed previously – the predominant overhead becomes the replication protocol. Interestingly, for a larger number of receivers (16 and 32), throughput converges to similar values across all combinations of envelope/cluster/block sizes. Whereas for larger envelope sizes this is due to the overhead of the replication protocol, for smaller envelope sizes this happens because the transmission of blocks to the receivers becomes the predominant overhead.

6.2.8 Geo-distributed Ordering Cluster

In addition to the aforementioned micro-benchmarks deployed in a local datacenter, we also conducted a geo-distributed experiment focused on collecting latency measurements at 4 frontends scattered across the Americas, with the nodes of the ordering service distributed all around the world: Oregon, Ireland, Sydney, and São Paulo (four BFT-SMaRt replicas), with Virginia standing as WHEAT's additional replica (five replicas). Since signatures generation requires considerable CPU power, we used instances of the type *m4.4xlarge*, with 16 virtual CPUs each. The frontends were deployed in Canada (frontend only), Oregon (collocated with leader node weighting V_{max} in WHEAT), Virginia (collocated with non-leader node, but still weighting V_{max}) and São Paulo. Each frontend was configured to launch enough client threads to keep node throughput always above 1000 transactions/second.

Results. Figure 6.14 presents the results for the geo-distributed micro-benchmark with a a block size of 10 envelopes. As expected, WHEAT's latency is consistently lower than BFT-SMaRt's across all frontends by almost 50%. It is worth pointing out that envelope size has a relatively minor impact on latency: across all regions, the difference between a 40 and a 4k bytes envelope was never above 29 milliseconds for any percentile or protocol. By



(a) Canada (clients only).

(b) Oregon (weighted V_{max}, leader node).

(c) Virginia (weighted V_{max}).

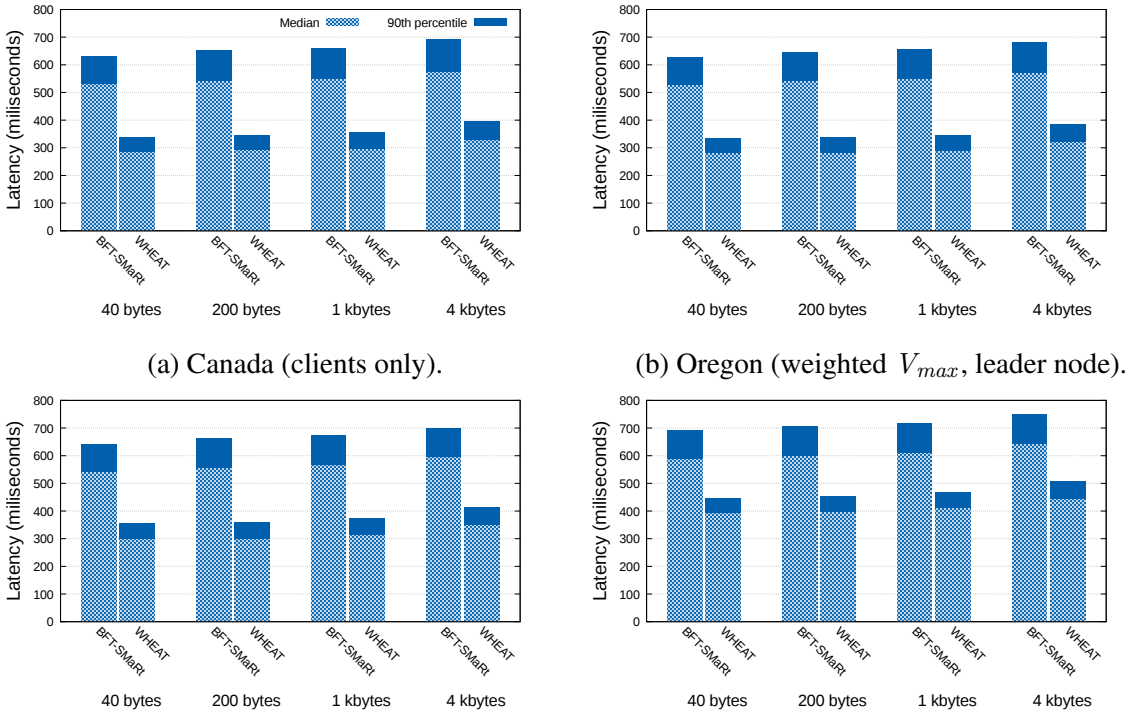
(d) São Paulo (weighted V_{min}).

Figure 6.14: Amazon EC2 latency results (4 receivers, blocks with 10 envelopes).

contrast, the placement of the frontends when using WHEAT exhibited a larger impact on latency: the difference between Virginia (weighted V_{max}) and São Paulo (weighted V_{min}) is above 43 milliseconds for BFT-SMaRt (+6.5%) and above 90 milliseconds (+23%) for WHEAT. In addition, the difference between São Paulo's and Oregon/Canada is even larger (58 milliseconds for BFT-SMaRt and 100 miliseconds for WHEAT, corresponding to an increase of +8,5% and +27% respectively). We also repeated the experiment for blocks of 100 envelopes (Figure 6.15). The pattern is similar to the previous configuration, but with increased latency (up to 63 milliseconds higher). This is because with similar workload but a larger block size, the rate of block generation decreases, which has a direct impact on latency.

6.2.9 Discussion

Our experimental evaluation shows that peak throughput is bound either by the rate at which block signatures are generated by a replica, or the rate of envelopes ordered by the total order protocol. Moreover, the results also suggest that, for smaller envelope sizes, increasing



(a) Canada (clients only).

(b) Oregon (weighted V_{max}, leader node).

(c) Virginia (weighted V_{max}).

(d) São Paulo (weighted V_{min}).

Figure 6.15: Amazon EC2 latency results (4 receivers, blocks with 100 envelopes).

the block size while decreasing the rate of signature generation can yield higher throughput than to simply rely on the maximum possible rate of signature generation. Nonetheless, for a higher number of repliers, throughput tends to converge to similar values across all micro-benchmarks. The obtained throughput are shown to be competitive with other production systems such as Symbiont Assembly, which is reported to reach 80.000 transactions/second with null data payloads on a Amazon EC2 setup of four nodes of type *c4.8xlarge*.[1] Additionally, even when transmitting blocks of 400 kbytes to 32 receivers in a cluster of 10 nodes, the ordering service still reaches a sustained throughput of approximately 2200 transactions/second – which is more than twice of Ethereum's theoretical peak of 1000 transactions/second (Buterin, 2016), and vastly superior than Bitcoin's peak of 7 transaction/second (Vukolić, 2015). Finally, latency measurements taken from a geo-replicated setting are also shown attractive, with values within half a second under moderate workload using WHEAT, even when accounting for large block sizes.

[1] https://symbiont.io/technology/assembly/

6.3 Additional Related Work

BFT transactional databases. Garcia Molina *et al.* (1986) proposed the earliest work to use state machine replication to tolerate Byzantine failures in the context of transactional databases. However, even thought this work enabled stricter isolation levels than SI, its sequential execution of transactions also provided limited performance.

In more recent years, there were other proposals for BFT transactional databases alongside Byzantium such as HRDB (Vandiver *et al.*, 2007), MITRA (Luiz *et al.*, 2014), Augustus (Padilha & Pedone, 2013) and Callinicos (Padilha *et al.*, 2016). HRDB is a commit barrier scheduling protocol that also provides Byzantine fault tolerance and supports concurrent transactions. This system provides high performance while still providing 1-copy serializability, but it relies on a trusted node to coordinate replicas. MITRA is a middleware similar to Byzantium that achieves 1-copy serializability and does not require multi-versioned DBMSs such as PostgreSQL, but the authors' evaluation of the system showcase lower performance than Byzantium. Finally, Augustus, and Callinicos are transactional key-value stores that tolerate Byzantine faults. Whereas Augustus relies on optimistic concurrency control akin MITRA and Byzantium, Callinicos proposes a novel conflict resolution protocol that is able to prevents conflicting transactions from aborting.

Distributed ledgers. The concept of blockchain was originally introduced by Bitcoin to solve the double spending problem associated with crypto-currency in permissionless peer-to-peer networks (Nakamoto, 2009). Since Bitcoin's inception and widespread adoption, other platforms based on Proof-of-Work blockchain have emerged. Within these new platforms, Ethereum is particularly relevant for its support of smart contracts (Wood, 2015).

Because of the known performance penalty associated with Proof-of-Work creation and the fact that Blockchain technology is gaining the attention of many industries, the idea of permissioned blockchains are quickly gaining traction. Examples of other permissioned blockchain platforms include Chain,[1] which uses the Federated Consensus algorithm.[2] Tendermint (Kwon, 2016) implements the BFT protocol designed by Buchman (2016). Kadena (Martino, 2016) uses a variant of the Raft consensus protocol (Ongaro & Ousterhout, 2014) adapted to Byzantine faults (Copeland & Zhong, 2014). Finally, Symbiont Assembly[3] uses a Go implementation of the Mod-SMaRt algorithm presented in Chapter 3 and heavily follows

[1] https://chain.com/
[2] https://chain.com/docs/1.2/protocol/papers/federated-consensus
[3] https://symbiont.io/technology/assembly/

the design of BFT-SMaRt. A recent survey compares all these permissioned protocols and points BFT-SMaRt as a prominent candidate for implementing this type of ledgers (Cachin & Vukolic, 2017b).

6.4 Concluding Remarks

In this final chapter we described the integration of our protocols into two practical systems. We began by presenting SteelDB, an implementation of the Byzantium protocol for replicated transactional databases and then proceeded to an ordering service created for the Hyperledger Fabric blockchain platform. Practical evaluations were performed for both these systems within a local cluster and in a geo-distributed environment. The evaluations revealed encouraging results for both systems. In the case of SteelDB, the system was shown to deliver acceptable performance for user interaction once we implemented speculative execution (local cluster) and integrated WHEAT into the codebase (geo-distributed). Meanwhile, our ordering service for Hyperledger Fabric displayed competitive throughput to other blockchain solutions such as Symbiont Assembly, Ethereum, and the original Bitcoin. It was also shown to display sub-second latency in a geo-distributed setting when powered by WHEAT.

7

Conclusions

This document describes the research conducted within the context of a PhD developed within the area of Byzantine fault tolerance, with a specialization on the state machine replication technique. This research explored methodologies aimed at achieving a higher level of robustness and performance for this type of replication. Our efforts began at the theoretical level with Mod-SMaRt, a modular and latency-optimal protocol that enables SMR to be more easily implemented and verifiable for correctness. Having Mod-SMaRt as our theoretical foundation, we then proceeded to implement the protocol into a working library dubbed BFT-SMaRt. This library was engineered to execute the protocol as efficiently as possible and maintained over the course of the whole PhD, with the objective of rendering the codebase as reliable as possible within the context of an academic endeavor. This accounts for the main contribution of the work presented, which was followed by an effort to optimize BFT-SMaRt for geo-distributed scenarios. The result of such effort was WHEAT, a variant of BFT-SMaRt that uses a novel vote assignment scheme that takes advantage of the heterogeneous nature of wide-area networks to provide better latency to clients. Towards the end of the thesis, we explored two use-cases for WHEAT: a BFT transactional middleware for relational databases and a BFT ordering service for a permissioned blockchain platform. The evaluation of both of these applications further indicates that within a geo-replicated setting, WHEAT reaches lower latency than BFT-SMaRt.

7.1 Impact

Prior to the beginning of the thesis, there was a lack of BFT SMR solutions that were both open-source and reliable. Even though a few open-source and generic codebases did exist (e.g., Castro & Liskov (2002); Clement *et al.* (2009a)), they were created for academic research and not continuously maintained over time. We posited that this situation caused an hindrance for researchers wishing to contribute to the area of BFT SMR with novel ideas, as well as other fields that may required replicated state machines as a building block. This observation led us to pursue the research direction explored in this thesis.

Our assessment was ultimately proven to have merit, given that over the course of the PhD we have noticed an increasing interest on BFT-SMART by our peers, as well as the emergence of many novel academic works that use BFT-SMART either as a building block or as codebase from which other system are derived from. At the time of this writing, the publications spawned from this thesis have a combined total of over 100 citations and used within the context of EU projects like TCLOUDS,[1] MASSIF,[2] SEGrid,[3] and SUPERCLOUD.[4] Due to to this thesis efforts at making the library stable and complete, the DepSpace coordination service (Bessani *et al.*, 2008) was rendered more resilient by association, which enabled that system to eventually be adopted as one of the building blocks for the Shared Cloud-backed File System (Bessani *et al.*, 2014) and later augmented to obtain the Extensible DepSpace model (Distler *et al.*, 2015). Besides these systems, BFT-SMART was also used as a building block for the FITCH platform (Cogo *et al.*, 2013), the SMaRtLight SDN controller (Botelho *et al.*, 2014), the MITRA transactional middleware (Luiz *et al.*, 2014), the SieveQ application firewall (Garcia *et al.*, 2016), a BFT Authentication and Authorization Infrastructure (Kreutz *et al.*, 2016) and an Intrusion-Tolerant SCADA control system (Nogueira *et al.*, 2017a). All these systems are rendered tolerant against Byzantine faults by integrating BFT-SMART into their designs. BFT-SMART also serves as a codebase that was modified and extended to create prototypes and deploy experimental evaluations for many papers. The library was adopted as basis to develop the CheapBFT system (Kapitza *et al.*, 2012), and more recently to serve as the control for the practical evaluation of the authors' COP architecture (Behl *et al.*, 2015). Due to the library's high-performance, the Hermes fault-injecting framework elected BFT-SMART as the target codebase to demonstrate the tool's capabilities (Martins *et al.*,

[1] http://www.tclouds-project.eu/
[2] http://www.massif-project.eu/
[3] http://www.segrid.eu/
[4] https://supercloud-project.eu/

2013). BFT-SMaRt was also used to create VFT-SMaRt, the prototype that adopts the Visigoth fault model (Porto *et al.*, 2015). Very recently, the library served as basis for a protocol that allows for on-the-fly partition transfer within replicated state machines (Nogueira *et al.*, 2017b), a work facilitated by the reconfiguration protocol that the library provides.

The aforementioned works belong all to the scope of academic research. However, the advent of blockchain technology also spawned new interest in Byzantine fault tolerance from for-profit organizations that are developing blockchain platforms, and BFT-SMaRt is being adopted by some of these projects. In particular, the open-source R3 Corda platform release a new version that provides a BFT distributed notary based on BFT-SMaRt.[1] Finally, Symbiont Assembly also uses a Go re-implementation of BFT-SMaRt protocol and architecture.[2]

7.2 Future Work

Due to the fact that the area of Byzantine fault tolerance is of high importance to distributed ledgers, we have steered our final research efforts towards the context of permissioned blockchains. Due to the modern interest by many organizations on this technology, we expect to follow up the research presented in the thesis within the context of geo-replication applied to permissioned blockchains.

As mentioned in the beginning of this chapter, this thesis focused on exploring techniques for achieving a robust and efficient deployment of replicated state machines. However, once we started applying our efforts on blockchain technology, we realized one pertinent direction of research not properly explored was *scalability*. This is because these systems are expected to operate with hundreds of nodes scattered around the world, rather than just with the configurations of 4 to 10 replicas used across the thesis (Vukolić, 2015). Any mid- to long-term work will be aimed at exploring methodologies to improve scalability in terms of number of nodes. Ideas currently being considered range from extending the Mod-SMaRt algorithm to support pipelining akin to the classic PBFT protocol (Castro & Liskov, 2002), implementing a propagation mechanism inspired by the well-known spanning tree protocol (Perlman, 1985), further research regarding partitioning of replicated state machines (Nogueira *et al.*, 2017b), and refurbishing BFT-SMaRt's reconfiguration protocol to support dynamic redistribution of WHEAT's vote assignment scheme.

[1] https://www.corda.net/2017/03/corda-m9-1-released/
[2] https://symbiont.io/technology/assembly/

Finally, the continued maintenance of BFT-SMART is an effort that shall remain as on-going. In particular, our intention is to fully integrate WHEAT's tentative execution and voting scheme mechanisms into the codebase, thus rendering WHEAT as the primary branch of development and support.

References

ABD-EL-MALEK, M., GANGER, G., GOODSON, G., REITER, M. & WYLIE, J. (2005). Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM SIGOPS Symposium on Operating Systems Principles*, Brighton, UK.

AGRAWAL, D. & ABADI, A.F. (1991). An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems*, **9**, 1–20.

AGUILERA, M. (2004). A pleasant stroll through the land of infinitely many creatures. *ACM SIGACT News*, **35**, 36–59.

AGUILERA, M.K., KEIDAR, I., MALKHI, D., MARTIN, J.P. & SHRAER, A. (2010). Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, **102**, 84–108.

AIYER, A.S., ALVISI, L., BAZZI, R.A. & CLEMENT, A. (2008). Matrix signatures: From macs to digital signatures in distributed systems. In *Proceedings of the 22nd Springer-Verlag International Symposium on Distributed Computing*, Arcachon, France.

ALSEBERG, P. & DAY, J. (1976). A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd IEEE International Conference on Software Engineering*, San Francisco, USA.

AMIR, Y. & WOOL, A. (1996). Evaluating quorum systems over the internet. In *Proceedings of the 26th IEEE International Symposium on Fault-Tolerant Computing*, Sendai, Japan.

AMIR, Y., DANILOV, C., DOLEV, D., KIRSCH, J., LANE, J., NITA-ROTARU, C., OLSEN, J. & ZAGE, D. (2010). Steward: Scaling Byzantine fault-tolerant replication to wide area networks. *IEEE Transactions on Dependable and Secure Computing*, **7**, 80–93.

AMIR, Y., COAN, B., KIRSCH, J. & LANE, J. (2011). Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, **8**, 564–577.

ANDERSON, J.C., LEHNARDT, J. & SLATER, N. (2010). *CouchDB: The Definitive Guide Time to Relax*. O'Reilly Media, Inc.

ANDROULAKI, E., BARGER, A., BORTNIKOV, V., CACHIN, C., CHRISTIDIS, K., CARO, A.D., ENYEART, D., FERRIS, C., LAVENTMAN, G., MANEVICH, Y., MURALIDHARAN, S., MURTHY, C., NGUYEN, B., SETHI, M., SINGH, G., SMITH, K., SORNIOTTI, A., STATHAKOPOULOU, C., VUKOLIC, M., COCCO, S.W. & YELLICK, J. (2018). Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the 13th ACM SIGOPS European Conference on Computer Systems*, Porto, Portugal.

ANKER, T., DOLEV, D., GREENMAN, G. & SHNAYDERMAN, I. (2003). Evaluating total order algorithms in WAN. In *Proceedings of the International Workshop on Large-Scale Group Communication*, Florence, Italy.

AUBLIN, P.L., MOKHTAR, S.B. & QUÉMA, V. (2013). RBFT: Redundant Byzantine fault tolerance. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, Philadelphia, PA, USA.

AUBLIN, P.L., GUERRAOUI, R., KNEŽEVIĆ, N., QUÉMA, V. & VUKOLIĆ, M. (2015). The next 700 BFT protocols. *ACM Transactions on Computer Systems*, **32**, 12:1–12:45.

BACK, A. (2002). Hashcash - a denial of service counter-measure. http://www.hashcash.org/papers/hashcash.pdf.

BAILIS, P., DAVIDSON, A., FEKETE, A., GHODSI, A., HELLERSTEIN, J.M. & STOICA, I. (2013). Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment*, **7**, 181–192.

BAKER, J., BOND, C., CORBETT, J.C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.M., LI, Y., LLOYD, A. & YUSHPRAKH, V. (2011). Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the 5th Biennial Conference on Innovative Data system Research*, Asilomar, California, USA.

BAKR, O. & KEIDAR, I. (2002). Evaluating the running time of a communication round over the internet. In *Proceedings of the 21st Symposium on Principles of Distributed Computing*, New York City, NY, USA.

BAKR, O. & KEIDAR, I. (2008). On the performance of quorum replication on the internet. Tech. Rep. UCB/EECS-2008-141, EECS Department, University of California, Berkeley.

BEHL, J., DISTLER, T. & KAPITZA, R. (2015). Consensus-oriented parallelization: How to earn your first million. In *Proceedings of the 16th Annual Middleware Conference*, ACM, New York City, NY, USA.

BEHL, J., DISTLER, T. & KAPITZA, R. (2017). Hybrids on steroids: SGX-based high performance BFT. In *Proceedings of the 12th ACM SIGOPS European Conference on Computer Systems*, ACM, New York City, NY, USA.

BEN-OR, M. (1983). Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proceedings of the 2rd ACM Symposium on Principles of Distributed Computing*, New York City, NY, USA.

BEN-OR, M., KELMER, B. & RABIN, T. (1994). Asynchronous secure computations with optimal resilience (extended abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, ACM, Los Angeles, CA, USA.

BENNETT, C. & TSEITLIN, A. (2012). Chaos monkey released in the wild. http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html.

BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E. & O'NEIL, P. (1995). A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, **24**, 1–10.

BESSANI, A. (2011). From Byzantine fault tolerance to intrusion tolerance (a position paper). In *Proceedings of the 5th Workshop on Recent Advances in Intrusion-Tolerant Systems*, Hong Kong, China.

BESSANI, A., SANTOS, M., FELIX, J., NEVES, N. & CORREIA, M. (2013). On the efficiency of durable state machine replication. In *Proceedings of the 2013 USENIX Annual Technical Conference*, San Jose, CA, USA.

BESSANI, A., MENDES, R., OLIVEIRA, T., NEVES, N., CORREIA, M., PASIN, M. & VERISSIMO, P. (2014). SCFS: A shared cloud-backed file system. In *Proceedings of the 2014 USENIX Annual Technical Conference*, Philadelphia, PA, USA.

BESSANI, A.N., ALCHIERI, E.P., CORREIA, M. & FRAGA, J.S. (2008). DepSpace: a Byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM European Systems Conference*, New York City, NY, USA.

REFERENCES

BEYER, B., JONES, C., PETOFF, J. & MURPHY, N. (2016). *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Incorporated.

BIRMAN, K., CHOCKLER, G. & VAN RENESSE, R. (2009). Toward a cloud computing research agenda. *SIGACT News*, **40**, 68–80.

BOKOR, P., KINDER, J., SERAFINI, M. & SURI, N. (2011). Efficient model checking of fault-tolerant distributed protocols. In *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks*, Hong Kong, China.

BOTELHO, F., BESSANI, A., RAMOS, F.M.V. & FERREIRA, P. (2014). On the design of practical fault-tolerant SDN controllers. In *Proceedings of the 2014 3rd European Workshop on Software Defined Networks*, Budapest, Hungary.

BRACHA, G. (1984). An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada.

BUCHMAN, E. (2016). *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. Master's thesis, University of Guelph.

BUDHIRAJA, N., MARZULLO, K., SCHNEIDER, F.B. & TOUEG, S. (1993). Distributed systems (2nd ed.). chap. The primary-backup approach, 199–216, ACM Press/Addison-Wesley Publishing Co.

BURROWS, M. (2006). The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, USA.

BUTERIN, V. (2015). Ethereum white paper. https://github.com/ethereum/wiki/wiki/White-Paper.

BUTERIN, V. (2016). Ethereum platform review: Opportunities and challenges for private and consortium blockchains. http://r3cev.com.

CACHIN, C. (2009). Yet another visit to Paxos. Tech. Rep. RZ 3754, IBM Research Zurich.

CACHIN, C. (2016). Architecture of the hyperledger blockchain fabric. https://www.zurich.ibm.com/dccl/papers/cachin_dccl.pdf.

CACHIN, C. & PORITZ, J.A. (2002). Secure intrusion-tolerant replication on the Internet. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks - DSN 2002*.

CACHIN, C. & TESSARO, S. (2005). Asynchronous verifiable information dispersal. In *Proceedings of the 19th International Conference on Distributed Computing*, Springer-Verlag, Cracow, Poland.

CACHIN, C. & VUKOLIC, M. (2017a). Blockchain consensus protocol in the wild (invited paper). In *Proceedings of 31th International Symposium on Distributed Computing*, Vienna, Austria.

CACHIN, C. & VUKOLIC, M. (2017b). Blockchain consensus protocols in the wild. Tech. Rep. arXiv:1707.01873, IBM Research - Zurich.

CACHIN, C., KURSAWE, K., PETZOLD, F. & SHOUP, V. (2001). Secure and efficient asynchronous broadcast protocols. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, London, UK.

CACHIN, C., KURSAWE, K. & SHOUP, V. (2005). Random oracles in constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology*, **18**, 219–246.

CANETTI, R. & RABIN, T. (1993). Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth ACM Symposium on Theory of Computing*, San Diego, CA, USA.

CARD, S.K., ROBERTSON, G.G. & MACKINLAY, J.D. (1991). The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York City, NY, USA.

CASON, D., MARANDI, P.J., BUZATO, L.E. & PEDONE, F. (2015). Chasing the tail of atomic broadcast protocols. In *2015 IEEE 34th IEEE Symposium on Reliable Distributed Systems*, Montreal, Quebec, Canada.

CASTRO, M. & LISKOV, B. (1999). Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, USA.

CASTRO, M. & LISKOV, B. (2002). Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, **20**, 398–461.

CASTRO, M., RODRIGUES, R. & LISKOV, B. (2003). BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems*, **21**, 236–269.

CHANDRA, T., GRIESEMER, R. & REDSTONE, J. (2007). Paxos made live - an engineering perspective. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, Portland, OR, USA.

CHANDRA, T.D. & TOUEG, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, **43**, 225–267.

CHUN, B.G., MANIATIS, P., SHENKER, S. & KUBIATOWICZ, J. (2007). Attested append-only memory: making adversaries stick to their word. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, Stevenson, WA, USA.

CLEMENT, A., KAPRITSOS, M., LEE, S., WANG, Y., ALVISI, L., DAHLIN, M. & RICHÉ, T. (2009a). UpRight cluster services. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, Big Sky, MT, USA.

CLEMENT, A., WONG, E., ALVISI, L., DAHLIN, M. & MARCHETTI, M. (2009b). Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, Boston, MA, USA.

COBBS, A.L. (2016). JSimpleDB: Language-driven persistence for java. https://cdn.rawgit.com/archiecobbs/jsimpledb/master/jsimpledb-language-driven.pdf.

COGO, V.V., NOGUEIRA, A., SOUSA, J., PASIN, M., REISER, H.P. & BESSANI, A. (2013). *FITCH: Supporting Adaptive Replicated Services in the Cloud*. Springer Berlin Heidelberg, Berlin, Heidelberg.

COPELAND, C. & ZHONG, H. (2014). Tangaroa: a Byzantine fault tolerant raft. http://www.scs.stanford.edu/14aucs244b/labs/projects/copeland_zhong.pdf.

CORBETT ET. AL, J.C. (2013). Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems*, **31**, 1–22.

CORREIA, M., NEVES, N.F. & VERÍSSIMO, P. (2006). From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *The Computer Journal*, **49**, 82–96.

CORREIA, M., FERRO, D.G., JUNQUEIRA, F.P. & SERAFINI, M. (2012). Practical hardening of crash-tolerant systems. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, USA.

COWLING, J., MYERS, D., LISKOV, B., RODRIGUES, R. & SHRIRA, L. (2006). HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of 7th Symposium on Operating Systems Design and Implementation - OSDI 2006*, Seattle, Washington.

DEAN, J. & BARROSO, L.A. (2013). The tail at scale. *Communications of the ACM*, **56**, 74–80.

DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P. & VOGELS, W. (2007). Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, New York City, NY, USA.

DISTLER, T., BAHN, C., BESSANI, A., FISCHER, F. & JUNQUEIRA, F. (2015). Extensible distributed coordination. In *Proceedings of the 10th ACM SIGOPS European Conference on Computer Systems*, Bordeaux, France.

DOBRE, D., MAJUNTKE, M., SERAFINI, M. & SURI, N. (2010). HP: Hybrid Paxos for WANs. In *Proceedings of the 2010 European Dependable Computing Conference*, Washington, DC, USA.

DOLEV, D. & HOCH, E.N. (2008). Constant-space localized Byzantine consensus. In *Proceedings of the 22nd EATCS international symposium on Distributed Computing*, Berlin, Heidelberg.

DOLEV, D., DIVISION, T.J.W.I.R.C.R. & STRONG, R. (1982). *Distributed Commit with Bounded Waiting*. IBM Thomas J. Watson Research Division.

DOUDOU, A., GARBINATO, B. & GUERRAOUI, R. (2005). *Dependable Computing Systems Paradigms, Performance Issues, and Applications*, chap. Tolerating Arbitrary Failures with State Machine Replication, 27–56. Wiley.

DUARTE, E., GARRETT, T., BONA, L., CARMO, R. & ZÜGE, A. (2010). Finding stable cliques of planetlab nodes. In *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks*, Chicago, IL, USA.

DUTTA, P., GUERRAOUI, R. & VUKOLIĆ, M. (2005). Best-case complexity of asynchronous Byzantine consensus. Tech. Rep. EPFL/IC/200499, School of Computer and Communication Sciences, EPFL.

DWORK, C. & NAOR, M. (1993). Pricing via processing or combatting junk mail. In *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, London, UK.

DWORK, C., LYNCH, N.A. & STOCKMEYER, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM*, **35**, 288–322.

EKWALL, R. & SCHIPER, A. (2007). Modeling and validating the performance of atomic broadcast algorithms in high latency networks. In *Proceedings of the 13th International Euro-Par Conference*, Rennes ,France.

ELISHA, S. & HAMILTON, J. (2014). Under the covers of AWS: Core distributed systems primitives that power our platform. In *AWS re:Invent*, Sands Expo, Las Vegas, NV, USA.

FISCHER, M.J., LYNCH, N.A. & PATERSON, M.S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, **32**, 374–382.

FITZPATRICK, B. (2004). Distributed caching with memcached. *Linux Journal*, **124**, 72–78.

FURLONGER, D. & VALDES, R. (2016). Hype cycle for blockchain technologies and the programmable economy. http://www.gartner.com/smarterwithgartner/3-trends-appear-in-the-gartner-hype-cycle-for-emerging-technologies-2016.

GARAY, J., KIAYIAS, A. & LEONARDOS, N. (2015). The bitcoin backbone protocol: Analysis and applications. In E. Oswald & M. Fischlin, eds., *Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Sofia, Bulgaria.

GARCIA, M., BESSANI, A., GASHI, I., NEVES, N. & OBELHEIRO, R. (2013). Analysis of operating systems diversity for intrusion tolerance. *Software - Practice and Experience*, **44**, 735–770.

GARCIA, M., NEVES, N. & BESSANI, A. (2016). SieveQ: A layered BFT protection system for critical services. *IEEE Transactions on Dependable and Secure Computing*, **PP**, 1–1.

GARCIA, R., RODRIGUES, R. & PREGUIÇA, N. (2011). Efficient middleware for Byzantine fault tolerant database replication. In *Proceedings of the 6th conference on Computer systems*, New York City, NY, USA.

GARCIA-MOLINA, H. & BARBARA, D. (1985). How to assign votes in a distributed system. *Journal of the ACM*, **32**, 841–860.

GARCIA MOLINA, H., PITTELLI, F. & DAVIDSON, S. (1986). Applications of Byzantine agreement in database systems. *ACM Transactions on Database Systems*, **11**, 27–47.

GIFFORD, D. (1979). Weighted voting for replicated data. In *Proceedings of the 7th ACM SIGOPS Symposium on Operating Systems Principles*, Pacific Grove, CA, USA.

GILBERT, S. & LYNCH, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, **33**, 51–59.

GILBERT, S. & LYNCH, N.A. (2012). Perspectives on the CAP Theorem. *Computer*, **45**, 30–36.

HADZILACOS, V. & TOUEG, S. (1993). Distributed systems (2nd ed.). chap. Fault-tolerant Broadcasts and Related Problems, 97–145, ACM Press/Addison-Wesley Publishing Co.

HERLIHY, M. & WING, J.M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems*, **12**, 463–492.

HUNT, P., KONAR, M., JUNQUEIRA, F. & REED, B. (2010). Zookeeper: Wait-free coordination for internet-scale services. In *Proceedings of the 2010 USENIX Annual Technical Conference*, Boston, MA, USA.

ISARD, M. (2007). Autopilot: Automatic data center management. *SIGOPS Operating Systems Review*, **41**, 60–67.

JOHNSON, D.B. & MENEZES, A.J. (1998). Elliptic curve DSA (ECSDA): An enhanced DSA. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, Berkeley, CA, USA.

JUNQUEIRA, F., MAO, Y. & MARZULLO, K. (2007). Classic Paxos vs Fast Paxos: Caveat emptor. In *Proceedings of the Workshop on Hot Topics in System Dependability*, Edinburgh, UK.

KAPITZA, R., BEHL, J., CACHIN, C., DISTLER, T., KUHNLE, S., MOHAMMADI, S.V., SCHRÖDER-PREIKSCHAT, W. & STENGEL, K. (2012). CheapBFT: Resource-efficient Byzantine fault tolerance. In *Proceedings of the 7th ACM SIGOPS European Conference on Computer Systems*, Bern, Switzerland.

KIHLSTROM, K.P., MOSER, L.E. & MELLIAR-SMITH, P.M. (2001). The SecureRing group communication system. *ACM Transactions on Information and System Security*, **4**, 371–406.

KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A. & WONG, E. (2009). Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems*, **27**, 45–58.

KRASKA, T., PANG, G., FRANKLIN, M.J., MADDEN, S. & FEKETE, A. (2013). MDCC: multi-data center consistency. In *Proceedings of the 8th ACM SIGOPS European Conference on Computer Systems*, Prague, Czech Republic.

KREUTZ, D., MALICHEVSKYY, O., FEITOSA, E., CUNHA, H., DA ROSA RIGHI, R. & DE MACEDO, D.D. (2016). A cyber-resilient architecture for critical security services. *Journal of Network and Computer Applications*, **63**, 173 – 189.

KWON, J. (2016). Tendermint: Consensus without mining, http://www.the-blockchain.com/docs/tendermint

LAKSHMAN, A. & MALIK, P. (2010). Cassandra: A decentralized structured storage system. *SIGOPS Operating Systems Review*, **44**, 35–40.

LAMPORT, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, **21**, 558–565.

LAMPORT, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems*, **16**, 133–169.

LAMPORT, L. (2001). Paxos made simple. *ACM SIGACT News*, **32**, 18–25.

LAMPORT, L. (2005). Generalized consensus and Paxos. Tech. Rep. MSR-TR-2005-33, Microsoft Research.

LAMPORT, L. (2006). Fast Paxos. *Distributed Computing*, **19**, 79–103.

LAMPORT, L., SHOSTAK, R. & PEASE, M. (1982). The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, **4**, 382–401.

LAMPORT, L., MALKHI, D. & ZHOU, L. (2010). Reconfiguring a state machine. *SIGACT News*, **41**, 63–73.

LAMPSON, B. (2001). The ABCD's of Paxos. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing*, Newport, RI, USA.

LI, H.C., CLEMENT, A., AIYER, A.S. & ALVISI, L. (2007). The Paxos register. In *Proceedings of the 26th IEEE Symposium on Reliable Distributed Systems*, Beijing, China.

LIU, S., VIOTTI, P., CACHIN, C., QUEMA, V. & VUKOLIC, M. (2016). XFT: Practical fault tolerance beyond crashes. In *12th USENIX Symposium on Operating Systems Design and Implementation*, SAVANNAH, GA, USA.

LORCH, J.R., ADYA, A., BOLOSKY, W.J., CHAIKEN, R., DOUCEUR, J.R. & HOWELL, J. (2006). The SMART way to migrate replicated stateful services. In *Proceedings of the 1st ACM SIGOPS European Conference on Computer Systems*, Leuven, Belgium.

LUIZ, A.F., LUNG, L.C. & CORREIA, M. (2014). MITRA: Byzantine fault-tolerant middleware for transaction processing on replicated databases. *SIGMOD Rec.*, **43**, 32–38.

LYNCH, N.A. (1996). *Distributed Algorithms*. Morgan Kauffman.

MALKHI, D. & REITER, M. (1998). Byzantine quorum systems. *Distributed Computing*, **11**, 203–213.

MAO, Y., JUNQUEIRA, F.P. & MARZULLO, K. (2008). Mencius: building efficient replicated state machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*.

MAO, Y., JUNQUEIRA, F.P. & MARZULLO, K. (2009). Towards low latency state machine replication for uncivil wide-area networks. In *In Workshop on Hot Topics in System Dependability*, Lisbon, Portugal.

MARIC, O., SPRENGER, C. & BASIN, D. (2015). Consensus refined. In *Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks*, Rio de Janeiro, Brazil.

MARTIN, J.P. & ALVISI, L. (2006). Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, **3**, 202–215.

MARTINO, W. (2016). Kadena: The first scalable, high performance private blockchain, http://kadena.io/docs/kadena-consensuswhitepaper-aug2016.pdf.

MARTINS, R., GANDHI, R., NARASIMHAN, P., PERTET, S., CASIMIRO, A., KREUTZ, D. & VERÍSSIMO, P. (2013). Experiences with fault-injection in a Byzantine fault-tolerant protocol. In D. Eyers & K. Schwan, eds., *Proceedings of the ACM/IFIP/USENIX 14th International Middleware Conference*, Beijing, China.

MILLER, A., XIA, Y., CROMAN, K., SHI, E. & SONG, D. (2016). The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Vienna, Austria.

MILLER, R.B. (1968). Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, ACM, San Francisco, CA, USA.

MILOSEVIC, Z., HUTLE, M. & SCHIPER, A. (2011). On the reduction of atomic broadcast to consensus with Byzantine faults. In *Proceedings of the 30th IEEE Symposium on Reliable Distributed Systems*, Madrid, Spain.

MILOSEVIC, Z., BIELY, M. & SCHIPER, A. (2013). Bounded delay in Byzantine-tolerant state machine replication. In *2013 IEEE 32th IEEE Symposium on Reliable Distributed Systems*, 61–70, Braga, Portugal.

MONIZ, H., NEVES, N.F., CORREIA, M. & VERISSIMO, P. (2011). RITAS: Services for randomized intrusion tolerance. *IEEE Transactions on Dependable and Secure Computing*, **8**, 122–136.

MORARU, I., ANDERSEN, D.G. & KAMINSKY, M. (2013). There is more consensus in egalitarian parliaments. In *Proceedings of 24th ACM SIGOPS Symposium on Operating Systems Principles*.

MOSTEFAOUI, A., MOUMEN, H. & RAYNAL, M. (2015). Signature-free asynchronous binary Byzantine consensus with T < n/3, O(n^2) messages, and O(1) expected time. *Journal of the ACM*, **62**, 31:1–31:21.

NAKAMOTO, S. (2009). Bitcoin: A peer-to-peer electronic cash system, http://bitcoin.org/bitcoin.pdf.

NAOR, M. & WOOL, A. (1998). Access control and signatures via quorum secret sharing. *Parallel and Distributed Systems, IEEE Transactions on*, **9**, 909–922.

NIELSEN, J. (1993). *Usability Engineering*. Morgan Kaufmann Publishers.

NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H.C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T. & VENKATARAMANI, V. (2013). Scaling memcache at facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, Lombard, IL, USA.

NOGUEIRA, A., BESSANI, A. & NEVES, N. (2017a). Intrusion-tolerant eclipse scada. In *Symposium on Innovative Smart Grid Cybersecurity Solutions*, Vienna, Austria.

NOGUEIRA, A., CASIMIRO, A. & BESSANI, A. (2017b). Elastic state machine replication. *IEEE Transactions on Parallel and Distributed Systems*, **PP**, 1–1.

OKI, B.M. & LISKOV, B. (1988). Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*.

ONGARO, D. & OUSTERHOUT, J. (2014). In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference*, Philadelphia, PA, USA.

PADILHA, R. & PEDONE, F. (2013). Augustus: Scalable and robust storage for cloud applications. In *Proceedings of the 8th ACM SIGOPS European Conference on Computer Systems*, Prague, Czech Republic.

PADILHA, R., FYNN, E., SOULÉ, R. & PEDONE, F. (2016). Callinicos: Robust transactional storage for distributed data structures. In *2016 USENIX Annual Technical Conference*, Denver, CO, USA.

PERLMAN, R. (1985). An algorithm for distributed computation of a spanningtree in an extended lan. *SIGCOMM Computer Communication Review*, **15**, 44–53.

PORTO, D., LEITÃO, J.A., LI, C., CLEMENT, A., KATE, A., JUNQUEIRA, F. & RODRIGUES, R. (2015). Visigoth fault tolerance. In *Proceedings of the 10th ACM SIGOPS European Conference on Computer Systems*, Bordeaux, France.

PÂRIS, J. (1986). Voting with witnesses: A consistency scheme for replicated files. In *In Proceedings of the 6th International Conference on Distributed Computing Systems*, Cambridge, MA, USA.

RABIN, M.O. (1983). Randomized Byzantine generals. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, Tucson, AZ, USA.

RANDELL, B., LEE, P. & TRELEAVEN, P.C. (1978). Reliability issues in computing system design. *ACM Computing Surveys*, **10**, 123–165.

REITER, M.K. (1994). Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communication Security*, Fairfax, VA, USA.

REITER, M.K. (1995). The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, 938, 99–110, Springer-Verlag.

REITER, M.K. (1996). A secure group membership protocol. *IEEE Transactions on Software Engineering*, **22**, 31–42.

RÜTTI, O., MILOSEVIC, Z. & SCHIPER, A. (2010). Generic construction of consensus algorithms for benign and Byzantine faults. In *Proceedings of the 40st IEEE/IFIP International Conference on Dependable Systems and Networks*, Chicago, IL, USA.

SAITO, Y. & SHAPIRO, M. (2005). Optimistic replication. *ACM Computing Surveys*, **37**, 42–81.

SANTOS, M.H.D. (2014). *Dependable data storage with state machine replication*. Master's thesis, Faculty of Sciences, University of Lisbon.

SANTOS, N. & SCHIPER, A. (2013a). Achieving high-throughput state machine replication in multi-core systems. In *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems*, Philadelphia, PA, USA.

SANTOS, N. & SCHIPER, A. (2013b). Optimizing Paxos with batching and pipelining. *Theoretical Computer Science*, **496**, 170–183.

SCHIPER, N., SUTRA, P. & PEDONE, F. (2009). Genuine versus non-genuine atomic multicast protocols for wide area networks: An empirical study. In *Proceedings of the 28th IEEE Symposium on Reliable Distributed Systems*, Niagara Falls, NY, USA.

SCHNEIDER, F. (1990). Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, **22**, 299–319.

SOUSA, P. (2006). Proactive resilience. In *Sixth European Dependable Computing Conference (EDCC-6) Supplemental Volume*, Coimbra, Portugal.

SZABO, N. (1996). Smart contracts: Building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought*.

TOUEG, S. (1984). Randomized Byzantine agreements. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, Vancouver, British Columbia, Canada.

TPC (2010). TPC Benchmark^{TM} C. http://www.tpc.org.

VANDIVER, B., BALAKRISHNAN, H., LISKOV, B. & MADDEN, S. (2007). Tolerating Byzantine faults in database systems using commit barrier scheduling. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, Stevenson, WA, USA.

VERÍSSIMO, P.E., NEVES, N.F. & CORREIA, M.P. (2003). Architecting dependable systems. chap. Intrusion-tolerant Architectures: Concepts and Design, 3–36, Springer-Verlag, Berlin, Heidelberg.

VERONESE, G., CORREIA, M., BESSANI, A. & LUNG, L.C. (2010). EBAWA: Efficient Byzantine agreement for wide-area networks. In *Proceedings of the 12th IEEE International High Assurance Systems Engineering Symposium*, San Jose, CA, USA.

VERONESE, G., CORREIA, M., BESSANI, A., LUNG, L.C. & VERISSIMO, P. (2013). Efficient Byzantine fault-tolerance. *IEEE Transactions on Computers*, **62**, 16–30.

VERONESE, G.S., CORREIA, M., BESSANI, A.N. & LUNG, L.C. (2009). Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proceedings of the 28th IEEE Symposium on Reliable Distributed Systems*, Niagara Falls, NY, USA.

VUKOLIĆ, M. (2010). The Byzantine empire in the intercloud. *ACM SIGACT News*, **41**, 105–111.

VUKOLIĆ, M. (2015). The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *Open Problems in Network Security - IFIP WG 11.4 International Workshop*, 112–125, Zurich, Switzerland.

VUKOLIĆ, M. (2017). Rethinking permissioned blockchains. In *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, Abu Dhabi, United Arab Emirates.

WALSH, L., AKHMECHET, V. & GLUKHOVSKY, M. (2009). RethinkDB - rethinking database storage. http://www.cs.toronto.edu/ ryanjohn/teaching/csc2531-f12/rethinkdb-whitepaper.pdf.

WARNS, T., STORM, C. & HASSELBRING, W. (2008). Availability of globally distributed nodes: An empirical evaluation. In *Proceedings of the 27th IEEE Symposium on Reliable Distributed Systems*, Naples, Italy.

WELSH, M., CULLER, D. & BREWER, E. (2001). SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM SIGOPS Symposium on Operating Systems Principles*, Chateau Lake Louise, Banff, Canada.

WOOD, G. (2015). Ethereum: A secure decentralised generalised transaction ledger, http://gavwood.com/paper.pdf.

YIN, J., MARTIN, J.P., VENKATARAMANI, A., ALVISI, L. & DAHLIN, M. (2003). Separating agreement form execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM SIGOPS Symposium on Operating Systems Principles*, Bolton Landing, NY, USA.

ZIELINSKI, P. (2004). Paxos at war. Tech. Rep. UCAM-CL-TR-593, University of Cambridge Computer Laboratory, Cambridge, UK.

Appendices

A

Mod-SMaRt correctness proof

In this Appendix we prove the correctness of Mod-SMaRt. In order to make such proof, two theorems need to be proven. The first theorem proves the safety of the protocol (i.e., it proves that all correct replicas process the same sequence of operations), whereas the second proves its liveness (i.e., that all operations sent by clients are eventually executed). However, before proving these theorems, it is useful declare and prove the three following lemmas.

Lemma A1: *All correct processes install the same sequence of regencies or a prefix of it.*
Proof: All correct replicas start their execution at regency 0. If no faults occur and the system remains synchronous, this is the regency all correct replicas will ever install, given that the algorithm never changes its regency in the normal case. However, a period of asynchrony or a faulty leader at the *VP-Consensus* primitive may trigger timeouts (lines 1-4 from Algorithm 3) which will lead a correct replica to ask for the installation of the next regency via the exchange of STOP messages (procedure $StartRegChange$). The algorithm guarantees that it is necessary at least one correct replica to start the process of changing the regency (line 14). Moreover, no correct replica installs a new regency without first receiving at least $2f + 1$ STOP messages (line 18), which means that (1) at least $f + 1$ correct replicas asked for the next regency to be installed; and (2) all correct replicas in the system will eventually receive enough STOP messages to justify the installation of the next regency. In addition, from procedure $StartRegChange$ and line 20, we can observe that no correct process install regency r without first having regency $r - 1$ installed. Therefore, all correct processes install the same sequence of regencies or a prefix of it. ■

Lemma A2: *Algorithm 3 enforces the* Termination *property of VP-Consensus.*
Proof: In order for the *Termination* property to be preserved within VP-Consensus, Algorithm 3 needs to guarantee that *VP-Timeout* is invoked as a result of expired timeouts, and across all correct processes in the system. From the algorithm, we can see that *VP-Timeout* is invoked after at least $2f + 1$ STOP messages associated to same regency are received. Given that a correct replica broadcast their STOP messages to all the others, this means that *VP-Timeout* will eventually be invoked at all correct replicas in the system. Furthermore, since $f + 1$ correct replicas had to broadcast STOP messages, at least one correct replica had to experience a timeout due to either lack of synchrony or a faulty leader within VP-Consensus. Furthermore, since timers are re-activated after *VP-Timeout* is invoked, the process is guaranteed to be repeated as many times as necessary. Hence, Algorithm 3 enforces the *Termination* property of VP-Consensus. ■

Lemma A3 *If a synchronization phase for regency g starts with a faulty leader l, then eventually a synchronization phase for regency $g' > g$ starts with correct leader $l' \neq l$.*
Proof: Each synchronization phase uses a special replica called 'leader', that receives at least $n - f$ STOPDATA messages and sends a single SYNC message to all replicas in the system (Algorithm 4, lines 26-31). The election of this leader is based on the current regency. Given that Lemma A1 proves that all correct replicas install the same sequence of regencies, this means that (1) an indefinite number of regency changes will take place; and (2) all correct replicas choose the same leader. If such leader is Byzantine, it can try to break from the protocol during this phase. But even if the leader is faulty, its behavior is constrained; it is not able to create false logs (because such logs are signed by the replicas that sent them in the STOPDATA messages). Additionally, each entry in the log contains the proof associated with each value decided in a consensus instance, which in turn prevents the replicas from providing incorrect decision values. Because of this, the worst a faulty leader can do, is:

1. *Not send the* SYNC *message to all replicas in the system*. The outcome if this situation depends on whether or not the leader sends the SYNC message to at least $n - f$ *cooperative* replicas (i.e., replicas that will follow the protocol, even if some of them are actually controlled by an adversary). If the leader does send the message to such replicas, the synchronization phase can finish, since there are $n - f$ replicas following the protocol. Otherwise, the timers associated with the operations waiting to be ordered will eventually be triggered in at least $f + 1$ correct replicas, which is enough to eventually restart the synchronization phase (Algorithm 3 and Lemma A1).

2. *Send two different* SYNC *messages to two different sets of replicas.* This situation can happen if the faulty leader waits for more than $n - f$ STOPDATA messages from replicas. The leader will then create sets of logs L and L', such that each set has exactly $n - f$ valid logs, and sends L to a set of replicas Q, and L' to Q'. In this scenario, Q and Q' may create different logs at line 38 of Algorithm 4 and resume normal phase at different consensus instances. But in order to make progress, at least $n - f$ correct replicas need to start the same consensus instance (because the consensus primitive needs these minimum amount of correct processes). Therefore, if the faulty leader does not send the same set of logs to a set Q_{n-f} with at least $n - f$ replicas that will follow the protocol, the primitive will not make progress. Hence, if the faulty leader wants to make progress, it has to send the same set of logs to at least $n - f$ cooperative replicas. Otherwise, timeouts will occur, and the synchronization phase is eventually restarted (Algorithm 3 and Lemma A1).

Finally, the new leader may be faulty again, but in that case, the same constraints explained previously will also apply to such leader. Because of this, when the system reaches a period of synchrony, after at most f regency changes, there there will be a new leader that is correct, and progress will be ensured. ■

Having the previous lemmas in mind, we now demonstrate that Mod-SMaRt upholds safety in the form of the following theorem:

Theorem A1 *Let r be the correct replica that executed the highest number of operations up to a certain instant. If r executed the sequence of operations $S = o_1, ..., o_s$, then all other correct replicas executed the same sequence of operations or a prefix of it.*

Proof: Assume that r and r' are two distinct correct replicas and o and o' are two distinct operations issued by correct client(s). Assume also b and b' to be the batch of operations where, respectively, o and o' were included upon being proposed in some consensus instance. For r and r' to be able not to execute the same sequence of operations or a prefix of it, at least one of three scenarios described below needs to happen.

1. *VP-Consensus instance i decides b in replica r, and decides b' in r'.* Since in this scenario the same sequence number can be assigned to 2 different batches, this will cause o and o' to be executed in different order at both r and r'. But by the *Agreement* and

Termination properties of VP-Consensus, such behavior is impossible; *Agreement* forbids two correct processes to decide differently, and *Termination* (preserved by Lemma A2) prevents any correct process from deciding more than once.

2. *b is a batch decided at VP-Consensus instance i in both r and r', but the operations in b are executed in different orders at r and r'.* This behavior can never happen because Algorithm 2 in line 13 forces the operations to be ordered deterministically, and since both r and r' are correct, the operations cannot be ordered using different criteria across correct replicas.

3. *Replica r' executes a sequence of operations containing gaps in relation to S.* From Algorithm 2, we can see that any operation is executed only after the *VP-Decide* event is triggered. This event is triggered either when a consensus instance decides a batch — which occurs during the normal phase — or when invoked by Algorithm 4 in line 45. In the absence of a synchronization phase, lines 3-6 of Algorithm 2 ensure that any consensus instance i is only started after instance $i-1$. This forces any correct process to execute the same sequence of operations without gaps.

   Lets now reason about the occurrence of a synchronization phase. In such case, r' may jump an to a consensus instance ahead of the one it was currently executing, effectively creating a gap on the sequence of instances it participates in. Nonetheless, Algorithm 4 still ensures r' will *VP-Decide* (and subsequently execute) all the decisions of the instances it skipped. This is because Algorithm 4 creates the L set (line 38) using batches from both the most up-to-date log contained in the SYNC message and from r' own $DecLog$. The algorithms also ensures L is obtained out of a log of valid operations (all satisfying $validDec$) that contains no gaps, i.e., satisfies the $noGaps$ predicate (lines 35-36). From Lemma A3, we know that there will eventually be a regency with a correct leader that supplies a valid log in the $SYNC$ message containing a log that will satisfy the aforementioned conditions. Algorithm 4 then sequentially triggers the *VP-Decide* event for each decision contained in L that has not been decided yet by r' (lines 44-45), thus forcing r' to execute a sequence of operations that is either S or a prefix of it.

Given that all the three aforementioned scenarios are impossible to occur, all correct replicas execute the sequence of operations S or a prefix of it. ■

Next lemmas are used to prove Mod-SMaRt' liveness in our system model (Theorem A2). Before presenting these lemmas, we need to state some required definitions. We say that an operation issued by a client c *completes* when c receives the same response for the operation from at least $\lceil \frac{n+f+1}{2} \rceil$ different replicas. We also consider that an operation sent by a client is *valid* if it is correctly signed, function $validCmd$ returns $TRUE$ for that operation, and if its sequence number is greater than the last sequence number of the last operation sent by that client.

Lemma A4 *If a correct replica receives a valid operation o, eventually all correct replicas receive o.*

Proof: We have to consider four possibilities concerning correct or faulty client and the synchrony of the system.

1. *Correct client and synchronous system.* In this case, the client will send its operation to all replicas, and all correct ones will receive the operation and store it in the $ToOrder$ set before a timeout occurs (Algorithm 2, line 1-2 plus procedure *RequestReceived*).

2. *Faulty client and synchronous system.* Assume a faulty client sends a valid operation o to at least one correct replica r. Such replica will initiate a timer t and start a consensus instance i (Algorithm 2, line 1-2 plus procedure *RequestReceived*). However, not enough replicas (less than $n - f$) will initialize a consensus instance i. Because of this, the timeout for t will eventually be triggered on the correct replicas that received it (Algorithm 3, line 1), and o will be propagated to all other replicas (line 2-3). From here, all correct ones will store the operation in the $ToOrder$ set (Algorithm 2, line 18-19 plus procedure *RequestReceived*).

3. *Correct client and asynchronous system.* In this case, a correct replica might receive an operation, but due to delays in the network, it will trigger its timeout before the client is able to reach all other replicas. Such timeout may be triggered in a correct replica and the message will be forwarded to other replicas. Because the client is correct, the operation will eventually be delivered to all correct replicas and each one will store it in the $ToOrder$ set.

4. *Faulty client and asynchronous system.* This case is similar to 3), with the addition that the client may send the request to as few as one correct replica. But like it was explained in 2), the replica will send the operation to all other replicas upon the first

timeout. This ensures that eventually the operation will be delivered to all correct replicas and each one will store it in the $ToOrder$ set.

Therefore, if a correct replica receives a valid operation o, them all correct replicas eventually receive o. ■

Lemma A5 *If one correct replica r starts consensus i, eventually $n - f$ replicas start i.*
Proof: We need to consider the client that issues the operations that are ordered by the consensus instance (correct or faulty), the replicas that start such instance (correct or faulty), and the state of the system (synchronous or asynchronous).

We can observe from Algorithm 2 that an instance is started after selecting a batch of operations from the $ToOrder$ set (lines 4-6). This set stores valid operations issued by clients. From Lemma A4, we know that a valid operation will eventually be received by all correct replicas, as long as at least one of those replicas receives it. Therefore, it is not necessary to consider faulty clients in this lemma.

When analyzing the protocol, we can verify that a consensus instance can be started, either during the normal phase (Algorithm 2, line 12), or at the end of the synchronization phase (Algorithm 4, lines 39-40). For both cases, we will prove that if at least one correct replica r starts a consensus instance i, at least $n - f$ replicas will also eventually start i.

The first case to consider is when *i is started by r for the first time during the normal phase.* Following this, there are two scenarios:

1. *r decides a value for i before a timeout is triggered.* For this scenario to happen, it is necessary that at least $n - f$ processes participated in the consensus instance without breaking the protocol. Therefore, $n - f$ replicas had to start instance i.

2. *A timeout is triggered before r is able to decide a value for i.* This situation can happen either because the system is passing through a period of asynchrony, or because of the presence of a faulty leader in the consensus instance (we are assuming that our primitive is leader driven). Let us consider a consensus instance j such that j is the highest instance started by a correct replica r'. Let us now consider the following possibilities:

   2-a) r started i and $i < j$. Remember that our algorithm executes a sequence of consensus instance, and no correct replica starts an instance without first deciding the previous one (Algorithm 2, lines 3-6). If $i < j$, j had to be started after i was

decided. But for i to have been decided, at least $n - f$ processes had to participate in the consensus instance without breaking the protocol. Therefore, $n - f$ replicas had to start instance i.

2-b) r started i and $i > j$. This situation is impossible, because if j is the highest instance started, and since both r and r' are correct, i cannot be higher than j.

2-c) r started i and $i = j$. In this case, the synchronization phase might be initialized before all correct replicas start i. Due to the period of asynchrony that triggered the synchronization phase, only f or less correct replica might have both decided $i - 1$ and started i. Hence, the log contained in the SYNC message may be one containing instances only up to $i - 2$ (received at Algorithm 4, line 32), even though there exist a log that goes up to $i - 1$. This is because, even if all replicas are correct, the leader can only safely waits for $n - f$ correct $\mathrm{STOPDATA}$ messages. Nonetheless, any correct replica that receives such log will either be already executing instance i or a previous one. Lines 39-40 ensure that any correct replica either jump to instance $i - 1$ or keep executing i, and because of the *Termination* property (ensured by Lemma A2) and lines 3-6 from Algorithm 2, any correct replica will start i after $i - 1$ is decided.

However, we still need to consider the correctness of the leader (correct or faulty), and the state of the system (synchronous or asynchronous). If the system is asynchronous, multiple synchronization phases might occur, where in each one a new leader will be elected. In each iteration, a faulty/malicious replica may be elected as leader and disrupt the behavior previously described; but from Lemma A3, we know that a faulty leader cannot prevent progress nor force replicas to process an invalid log. Therefore, when the system finally becomes synchronous, eventually a correct leader will be elected, and either i or $i - 1$ are started by $n - f$ replicas.

The second case to consider is when *r starts i for the first time at the end of a synchronization phase.* If r is starting i at this point in the protocol, it is because i is not the consensus instance that it was currently executing; r received a decision for $i - 1$ in the log contained in the SYNC message, thus starting i based in such log. Nonetheless, this is a situation equivalent to scenario 2-c. Therefore, eventually $n - f$ replicas will start i. ■

Theorem A2 *A valid operation requested by a client eventually completes.*
Proof: Let o be a valid operation which is sent by a client, and I the finite set of consensus instance where o is proposed. Due to Lemma A4, we know that o will eventually be received

by all correct replicas, and at least one of them will propose o in at least one instance of I (because the *fair* predicate ensures this). By Lemma A5, we also know that such instances will eventually start in $n - f$ replicas.

Furthermore, let us show that there must be a consensus instance $i \in I$ where o will be part of the batch that is decided in i. As already proven in Lemma A4, all correct replicas will eventually receive o. Second, we use the *fair* predicate to avoid starvation, which means that any operation that is yet to be ordered, will be proposed again. Because of this, all correct replicas will eventually include o in a batch of operations for the same consensus instance i. Furthermore, the γ predicate of our *external validity* property ensures that 1) the operations in the batch sent by the consensus leader is not empty 2) it is correctly signed, and 3) each sequence number of each operation is the next sequence number expected from the client that issued it.

Because there are enough replicas starting i (due to Lemma A5) and the *Termination* property of consensus will hold (due to Lemma A2), the consensus instance will eventually decide a batch containing o in $\lceil \frac{n+f+1}{2} \rceil$ correct replicas. Finally, these same replicas will send a *REPLY* message to the client (Algorithm 2 at line 17), notifying it that the operation o was ordered and executed. Therefore, a valid operation requested by a client eventually completes. ■

B

VP-Consensus algorithm

In this appendix we discuss how a *VP-Consensus* primitive can be implemented from a standard consensus algorithm that already provides the three classic properties mentioned in Section 3.4 (*Termination*, *Integrity* and *Agreement*). More precisely, we explain how the Byzantine algorithm described in (Cachin, 2009) can be extend to implement *VP-Propose*, *VP-Decide* and *VP-Timeout*, as well as to satisfy the *External Validity* and *External Provability* properties.

B.1 Algorithm

Cachin's Byzantine algorithm is a leader-based consensus abstraction (*lc*), that uses multiple modules to provide *Termination*, *Integrity* and *Agreement*. Specifically, these modules are called *epoch* and *epoch-change*.[1] The module *epoch* (*ep*) enforces the *Integrity* and *Agreement* properties, by enabling processes to propose values and potentially decide a value. By contrast, *epoch-change* (*ec*) enforces *Termination* by creating and aborting epochs until a value is eventually decided in one.

Figure B.1 illustrates an execution of (Cachin, 2009), which is comprised by a sequence of epochs *ep*. The message pattern employed by *ep* is depicted in Figure B.2, which generally requires 5 communication steps (Figure B.2a), but can be reduce to 3 steps during the first epoch for any instance of *lc* (Figure B.2b).

[1] Depending of the implementations of these two modules, the consensus algorithm can either withstand crash or Byzantine faults. We focus on the implementations for the Byzantine fault model.

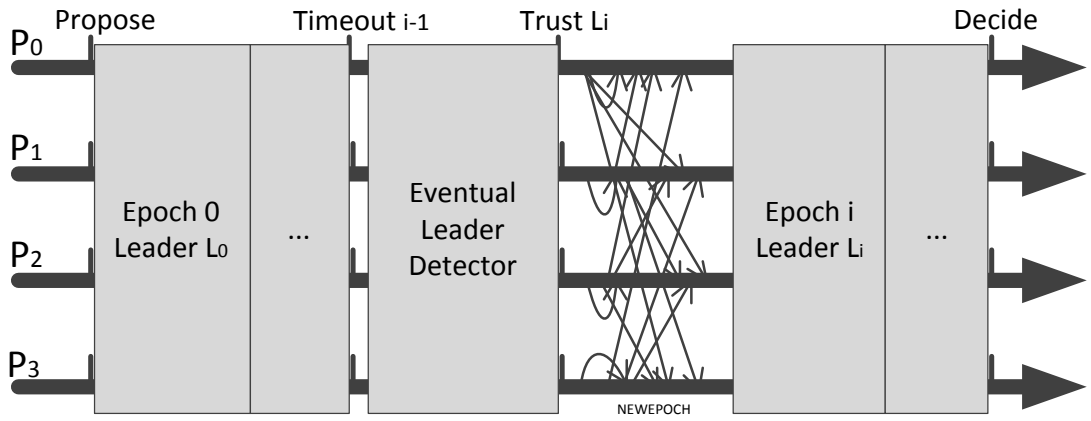
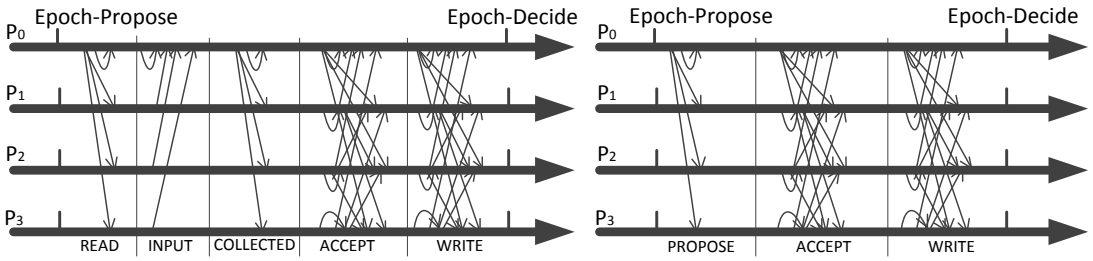


Figure B.1: Byzantine leader-driven consensus.



(a) Standard pattern ($i > 0$). (b) Optimized pattern ($i = 0$).

Figure B.2: Epoch message pattern.

Epoch-change is responsible for managing the sequence of epochs related to an *lc* instance, by assigning epoch i to a leader l_i. If an epoch is unable to finish before a timeout is triggered, epoch-change complains to an *eventual leader detector* oracle (Ω) (Chandra & Toueg, 1996). Once each process is notified to trust a different leader by Ω, it broadcasts a *NEWEPOCH* message and waits for other processes to also broadcast this message. Once enough messages are received, epoch i is aborted and epoch $i + 1$ is initialized with leader l_{i+1}.

To obtain the VP-Consensus primitive from *lc*, it is necessary to modify *lc* and *ep* as illustrated in Algorithms 5-6. Upon initializing the algorithm (lines 1–10), the leader for *ep* is appointed by the application that invokes *VP-Propose* instead of being selected by the algorithm. Furthermore, each VP-Consensus instance is uniquely identified by $consID$.

Algorithm 5: VP-Consensus implementation derived from Cachin (2009) (part 1).

```
   // Replaces ⟨Init⟩ in Abstract Leader-Based Consensus
1  Upon VP-Propose(i, l, γ, v) do
2      consID ← i
3      proposed ← FALSE
4      decided ← FALSE
5      Obtain initial timestamp ets_0
6      Initialize an instance of Epoch-change ec with ID i
7      Initialize a new instance of Epoch ep with ID i, timestamp ets_0,
8          leader l, predicate γ and state ⟨0, ⊥⟩
9      state ← ⟨ets_0, l⟩
10     val ← v

   // Replaces ⟨T.timeout⟩ in Abstract Leader-Based Consensus
11 Upon VP-Timeout(consID, l) do
12     if decided = FALSE
13         Trigger ⟨Ω.trust | l⟩

   // Replaces ⟨ep.decide | v⟩ in
   // Abstract Leader-Based Consensus
14 Upon ⟨ep.decide | (v, Γ)⟩ do
15     if decided = FALSE
16         decided ← TRUE
17         Trigger VP-Decide(consID, v, Γ)

   // Replaces ⟨ep.propose | v⟩ in Epoch with
   // Byzantine faults (only leader l)
18 Upon ⟨ep.propose | v⟩ do
19     if val =⊥
20         val ← v
21     if ets = ets_0
22         Trigger ⟨abeb.broadcast | [PROPOSE, v]⟩
23     else
24         Trigger ⟨abeb.broadcast | [READ]⟩
```

Algorithm 6: VP-Consensus implementation derived from Cachin (2009) (part 2).

```
   // Additional event for Epoch with Byzantine faults
25 Upon ⟨abeb.deliver | p_j, [PROPOSE, v]⟩ do
26     if ets = ets_0
27         writeset ← {(ets, v)}
28         Trigger ⟨abeb.broadcast | [WRITE, v]⟩

   // Modifies generation of WRITE messages for
   // Epoch with Byzantine faults
29 Upon ∃ v such that |{j|written[j] = v}| > (n+f)/2 do
30     (valts, val) ← (ets, v)
31     written ← [⊥]^n
32     σ ← sign(self, ACCEPT||self||v)
33     Trigger ⟨abeb.broadcast | [ACCEPT, v, σ]⟩

   // Modifies generation of ACCEPT messages for
   // Epoch with Byzantine faults
34 Upon ⟨abeb.deliver | j, [ACCEPT, v, σ]⟩ do
35     accepted[j] ← [ACCEPT, v, σ]

   // Modifies decision generation for
   // Epoch with Byzantine faults
36 Upon ∃ v such that |{j|accepted[j] = [ACCEPT, v, ∗]}| > (n+f)/2 ∧ γ(v) = TRUE
   do
37     Trigger ⟨ep.decide|(v, accepted)⟩
38     halt
```

Instead of having processes complain to Ω upon an internal timeout, processes expect the application to invoke *VP-Timeout* to force Ω to immediately appoint a new trusted leader (lines 11–13). This can be done because correct processes must only invoke *VP-Timeout* after a new leader is elected.

To enforce *External Validity*, we make the algorithm verify the committed value using predicate γ (line 36). To ensure *External Provability*, each process sign its own *ACCEPT* message (lines 29–33), store signatures from *ACCEPT* messages sent from all replicas (lines 34–35), and *VP-Decide* the decision value alongside the collected messages (lines 14–17, 36–38). In addition, we also implement the optimization that enables the protocol to decided a value within three communication steps in the first epoch (lines 18–24).

B.2 Correctness

To prove that our extension is correct, we show that (1) our modifications preserve the original properties of the protocol, and (2) the extension makes the algorithm satisfy the *External Validity* and *External Provability* properties. Moreover, since the aforementioned optimization is already discussed in Cachin (2009), we abstain from proving its correctness in the theorems bellow.

Theorem B1 *For any VP-Consensus instance c, if all correct processes (1) eventually appoint a correct leader l to c; and (2) propose values v_i such that $\forall i,\ \gamma(v_i) = \text{TRUE}$, the algorithm still enforces* Termination, Integrity *and* Agreement.

Proof: Correct processes can appoint a leader to the consensus primitive using *VP-Propose* (lines 1–10) and *VP-Timeout* to force the algorithm to trust a different process (lines 11–13). If all correct processes eventually appoint a correct leader l to c, they enforce the properties of the eventual leader detector Ω that is required by the original algorithm.

Line 36 prevents a correct process from deciding a value v if $\gamma(v) = \text{FALSE}$. This can happen if an epoch is appointed a faulty leader that proposes v. However, as long as all correct processes enforce Ω, eventually a new epoch will be started with a correct leader which proposes value v' such that $\gamma(v') = \text{TRUE}$. Given that all correct processes propose a value that respects predicate γ, such value must exist and will eventually become the decision.

Finally, the generation of Γ (lines 29–35) and respective delivery (lines 14–17, 36–38) only adds additional information to be delivered alongside the decided value, which does not result in any alteration to the original protocol. Hence, if all correct processes implement Ω using *VP-Propose*/*VP-Timeout* and those processes propose a valid value, the algorithm still enforces its original properties (*Termination*, *Integrity* and *Agreement*). ■

Theorem B2 *Algorithms 5-6 satisfy* External Validity.

Proof: Line 36 prevents a correct process from deciding a value v if $\gamma(v) = \text{FALSE}$, hence any value that is decided must satisfy predicate γ. From theorem B1, we know that such value will eventually be decided. Hence, our extension provides *External Validity*. ■

Theorem B3 *Algorithms 5-6 satisfy* External Provability.

Proof: Lines 29–35 produce signatures for the *ACCEPT* messages exchange among processes. These messages and respective signatures are then delivered alongside the decision

obtained in the consensus instance (lines 14–17, 36–38). Since signatures are assumed to be unforgeable, processes can validate the *ACCEPT* messages to verify if a quorum of processes did produce the messages. Since a quorum of signed *ACCEPT* messages comprise the last communication step before a decision is delivered, they can be used to prove if a value v was indeed decided in a consensus instance c. Hence, our extension provides *External Provability*. ■

C

WHEAT vote assignment scheme correctness proof

In this appendix, we present proofs of correctness for the properties of the vote assignment schemes presented in Section 5.3.2.

C.1 Preliminary Definitions

We start by establishing the following definitions:

Definition 1. *A subset that contains all replicas holding V_{max} votes is called a MAX subset.*

Definition 2. *A subset that contains all replicas holding V_{min} votes is called a MIN subset.*

Note that the above definitions do not prohibit a MAX subset to contain V_{min} replicas, or a MIN subset to contain V_{max} replicas.

C.2 CFT vote assignment

Theorem C1 (Safe minimality): *There exists at least one minimal quorum in the system.*
Proof: To satisfy this property, the sum of all votes in a minimal quorum must be $F_v + 1$ (the CFT value of Q_v). In addition, the minimal quorum must also be comprised by $f + 1$ replicas.

By equation (5.5), we know that f replicas holding V_{max} votes equals F_v:

$$fV_{max} = F_v$$

The above expression already accounts for a MAX subset, i.e, all the f replicas holding V_{max} votes. However, we can see that these f replicas are not enough to reach $F_v + 1$ votes. To reach this value, we need to add one of the replicas holding V_{min} votes. Since we assume that $V_{min} = 1$, we can expand the previous expression as follows:

$$fV_{max} + V_{min} = F_v + 1$$

This gives us the sum of votes provided by a minimal quorum under CFT mode (i.e., $Q_v = F_v + 1$). Furthermore, it also shows that any minimal quorum is comprised by $f + 1$ replicas. Hence, there exists at least one minimal quorum in the system. ■

Theorem C2 (Availability): *There is always a quorum available in the system that holds Q_v votes.*
Proof: Since f replicas may fail by crash or be too slow, the quorum may need at most $n - f$ replicas. From equation (5.1) we can unpack:

$$n - f = (2f + \Delta + 1) - f = f + \Delta + 1$$

Let us assume the worst case scenario, where all of the f replicas holding V_{max} are not present in a subset of $n - f$ replicas. This is the scenario that will subtract the most from the sum of all votes, leaving the system with an MIN subset comprised strictly by the V_{min} replicas. In this case, we can expand the previous expression to infer the number of votes as follows:

$$f + \Delta + 1 \Rightarrow (f + \Delta + 1)V_{min}$$

Using equation (5.5), we have:

$$(f + \Delta + 1)V_{min} = (f + \Delta)V_{min} + V_{min} = F_v + 1$$

Given that we considered the worst case scenario, any other combination of hosts will hold a sum of votes equal or greater than $Q_v = F_v + 1$. Hence, there is always a quorum available in the system that holds Q_v votes. ■

Before proving the Consistency property, we start by presenting the following lemmas:

Lemma C1: *All MAX subsets intersect by at least one replica.*
Proof: Any MAX subset contains all f replicas holding V_{max} votes. Therefore, any MAX subset will share all these f replicas among them. Since $f \geq 1$, MAX subsets always intersect by at least one replica. ■

Lemma C2: *All MIN subsets intersect by at least one replica.*
Proof: Any MIN subset contains all $f + \Delta + 1$ replicas holding V_{min} votes. Therefore, any MIN subset will share all these $f + \Delta + 1$ replicas among them. Since $f + \Delta + 1 \geq 1$, MIN subsets always intersect by at least one replica. ■

Lemma C3: *If a subset holds $Q_v = F_v + 1$ votes, it is either a MAX or a MIN subset.*
Proof: This lemma is proved by contradiction. Lets attempt to obtain $Q_v = F_v + 1$ votes from a subset comprised by all but one V_{max} replica and all but one V_{min} replica:

$$\begin{aligned} Q_v &= (f-1)V_{max} + (\Delta + f)V_{min} \Rightarrow \\ F_v + 1 &= fV_{max} - V_{max} + (\Delta + f)V_{min} \end{aligned}$$

By equation (5.5), we can convert the above expression to:

$$\begin{aligned} F_v + 1 &= F_v - V_{max} + \Delta + f \Rightarrow \\ 1 &= -V_{max} + \Delta + f \end{aligned}$$

By equation (5.8), we can convert the above expression to:

$$1 = -(1 + \frac{\Delta}{f}) + (\Delta + f) \Rightarrow \Delta + f = 2 + \frac{\Delta}{f}$$

To solve the equation above, we can either assume that $\Delta = 2$, $f = \frac{\Delta}{f}$ or $\Delta = \frac{\Delta}{f}$, $f = 2$. We thus use two equation systems to find a solution.

$$(a) \begin{cases} \Delta + f = 2 + \frac{\Delta}{f} \\ \Delta = \frac{\Delta}{f} \\ f = 2 \end{cases}$$

$$(b) \begin{cases} \Delta + f = 2 + \frac{\Delta}{f} \\ \Delta = 2 \\ f = \frac{\Delta}{f} \end{cases}$$

If we develop Δ from system (a), we obtain:

$$\Delta = \frac{\Delta}{f} \Rightarrow f = 1$$

However, system (a) already has $f = 2$, which leads to a contradiction. This leaves us with system (b), where we can develop f as follows:

$$f = \frac{\Delta}{f} \Rightarrow f^2 = \Delta$$

Since $\Delta = 2$, it follows that:

$$f^2 = 2$$

However, f must be a natural number, since it represents the maximum amount of replicas that can fail. Therefore, given our system model, this solution is invalid.

Finally, we cannot consider even less V_{max} or V_{min} replicas, since the sum of votes decreases even more from Q_v. On the other hand, if we assume more replicas, we are rendering the set either MAX or MIN. Hence, any subset that holds Q_v votes must be either MAX or MIN. ■

We now state the following corollaries:

Corollary C1: *If a MAX subset holds $Q_v = F_v + 1$ votes, it contains $f + 1$ replicas.*
Proof: From Safe minimality, a minimal quorum holds $Q_v = F_v + 1$ votes and contains $f + 1$ replicas. In addition, the correspondent proof shows that a minimal quorum contains all V_{max} replicas, which makes it a MAX subset. ■

Corollary C2: *If a MIN subset holds $Q_v = F_v + 1$ votes, it contains $n - f$ replicas.*
Proof: From Availability, it is always possible to access a quorum holding $Q_v = F_v + 1$ votes with at most $n - f$ replicas. In addition, the correspondent proof shows that such quorum contains all V_{min} replicas, which makes it a MIN subset. ■

Now we use Lemmas C1-C3 and Corollaries C1-C2 to prove the following theorem:

Theorem C3: *All subsets holding $Q_v = F_v + 1$ votes are quorums that intersect in at least one replica.*
Proof: From Lemma C3, we can infer that any possible quorum can be classified as either a MAX or MIN subset. We already showed that MAX (resp. MIN) subsets intersect in at least one replica in Lemma C1 (resp. Lemma C2). To finish proving this theorem, we need to show that, as long as they hold $Q_v = F_v + 1$ votes, any MAX and MIN intersect in at least one replica. From Corollary C1, any MAX that is a minimal quorum (i.e., holds Q_v votes) contains $f + 1$ replicas. From Corollary C2, any MIN holding Q_v votes is comprised by $n - f$ replicas. Therefore, if we sum the number of replicas contained in MAX and MIN, we have:

$$\begin{aligned}(f+1) + (n-f) = \\ f + 1 + 2f + \Delta + 1 - f = \\ 2f + \Delta + 2\end{aligned}$$

Since $2f + \Delta + 2 \geq n$, MAX and MIN intersect in at least one replica. Therefore, all subsets holding $Q_v = F_v + 1$ votes are quorums that intersect in at least one replica. ■

We can finally prove the Consistency property using Theorem C3:

Theorem C4 (Consistency): *All quorums that hold Q_v votes intersect by at least one correct replica.*
Proof: Theorem C3 shows that gathering Q_v votes guarantee quorum intersection in one replica. Since replicas can only fail by crash, the replica in the intersection must be correct. Therefore, if Q_v votes are gathered, we have quorums that intersect by at least one correct replica. ■

C.3 BFT vote assignment

Theorem C5 (Safe minimality): *There exists at least one minimal quorum in the system.*

Proof: To satisfy this property, the sum of all votes in a minimal quorum must be $2F_v + 1$ (the BFT value of Q_v). In addition, the minimal quorum must also be comprised by $2f + 1$ replicas.

As we concluded in Section 5.3.2, there must exist $2f$ replicas holding V_{max} votes under BFT mode. Furthermore, by equation (5.5), we know that f replicas holding V_{max} votes equals F_v. Based on these two observations, we can unpack the expression:

$$2fV_{max} = 2F_v$$

The above expression already accounts for a MAX subset, i.e, all the $2f$ replicas holding V_{max} votes. However, we can see that these $2f$ replicas are not enough to reach $2F_v + 1$ votes. To reach this value, we need to add one of the replicas holding V_{min} votes. Since we assume that $V_{min} = 1$, we can expand the previous expression as follows:

$$2fV_{max} + V_{min} = 2F_v + 1$$

The above expression gives us the sum of votes provided by a minimal quorum under BFT mode (i.e., $Q_v = 2F_v + 1$). Furthermore, it also shows that any minimal quorum is comprised by $2f + 1$ replicas. Hence, there exists at least one minimal quorum in the system. ■

Theorem C6 (Availability): *There is always a quorum available in the system that holds Q_v votes.*

Proof: Since f replicas may fail by crash or refuse to answer a request, the quorum may need at most $n - f$ replicas. From equation (5.1)[1] we can unpack:

$$n - f = (3f + \Delta + 1) - f = 2f + \Delta + 1$$

Let us assume the worst case scenario, where f replicas holding V_{max} are not present in a subset of $n - f$ replicas. This is the scenario that will subtract the most from the sum of all votes, leaving the system with an MIN subset. However, since the total amount of V_{max} replicas in the system is $2f$, there are still other f V_{max} replicas that did not fail. In this case, we can expand the previous expression to infer the number of votes as follows:

[1] Under BFT mode, equation (5.1) becomes $n = 3f + 1 + \Delta$

$$2f + \Delta + 1 = f + f + \Delta + 1 \Rightarrow fV_{max} + (f + \Delta + 1)V_{min} \\ = fV_{max} + (f + \Delta)V_{min} + 1$$

Using equation (5.5), we have:

$$F_v + F_v + 1 = 2F_v + 1$$

Given that we considered the worst case scenario, any other combination of hosts will hold a sum of votes equal or greater than $Q_v = 2F_v + 1$, since they will contain at least one replica with V_{max} votes. Hence, there is always a quorum available in the system that holds Q_v votes. ■

Before proving the Consistency property, we start by presenting the following lemmas:

Lemma C4: *All MAX subsets intersect by at least $f + 1$ replicas.*

Proof: Any MAX subset contains all $2f$ replicas holding V_{max} votes. Therefore, any MAX subset will share all these $2f$ replicas among them. Since $2f \geq f + 1$, MAX subsets always intersect by at least $f + 1$ replicas. ■

Lemma C5: *All MIN subsets intersect by at least $f + 1$ replicas.*

Proof: Any MIN subset contains all $f + \Delta + 1$ replicas holding V_{min} votes. Therefore, any MIN subset will share all these $f + \Delta + 1$ replicas among them. Since $f + \Delta + 1 \geq f + 1$, MIN subsets always intersect by at least $f + 1$ replicas. ■

Lemma C6: *If a subset holds $Q_v = 2F_v + 1$ votes, it is either a MAX or a MIN subset.*

Proof: This lemma is proved by contradiction. Lets attempt to obtain $Q_v = 2F_v + 1$ votes from a subset comprised by all but one V_{max} replica and all but one V_{min} replica:

$$Q_v = (2f - 1)V_{max} + (\Delta + f)V_{min} \Rightarrow \\ 2F_v + 1 = 2fV_{max} - V_{max} + (\Delta + f)V_{min}$$

By equation (5.5), we can convert the above expression to:

$$2F_v + 1 = 2F_v - V_{max} + \Delta + f \Rightarrow$$
$$1 = -V_{max} + \Delta + f$$

By equation (5.8), we can convert the above expression to:

$$1 = -(1 + \frac{\Delta}{f}) + (\Delta + f) \Rightarrow \Delta + f = 2 + \frac{\Delta}{f}$$

To solve the equation above, we can either assume that $\Delta = 2$, $f = \frac{\Delta}{f}$ or $\Delta = \frac{\Delta}{f}$, $f = 2$. We thus use two equation systems to find a solution.

$$(c) \begin{cases} \Delta + f = 2 + \frac{\Delta}{f} \\ \Delta = \frac{\Delta}{f} \\ f = 2 \end{cases}$$

$$(d) \begin{cases} \Delta + f = 2 + \frac{\Delta}{f} \\ \Delta = 2 \\ f = \frac{\Delta}{f} \end{cases}$$

If we develop Δ from system (c), we obtain:

$$\Delta = \frac{\Delta}{f} \Rightarrow f = 1$$

However, system (c) already has $f = 2$, each leads to a contradiction. This leaves us with system (d), where we can develop f as follows:

$$f = \frac{\Delta}{f} \Rightarrow f^2 = \Delta$$

Since $\Delta = 2$, it follows that:

$$f^2 = 2$$

However, f must be a natural number, since it represents the maximum amount of replicas that can fail. Therefore, given our system model, this solution is invalid.

Finally, we cannot consider even less V_{max} or V_{min} replicas, since the sum of votes decreases even more from Q_v. On the other hand, if we assume more replicas, we are rendering

the set either MAX or MIN. Hence, any subset that holds Q_v votes must be either MAX or MIN. ■

We now state the following corollaries:

Corollary C3: *If a MAX subset holds $Q_v = 2F_v + 1$ votes, it contains $2f + 1$ replicas.*
Proof: From Safe minimality, a minimal quorum holds $Q_v = 2F_v + 1$ votes and contains $2f + 1$ replicas. In addition, the correspondent proof shows that a minimal quorum contains all V_{max} replicas, which makes it a MAX subset. ■

Corollary C4: *If a MIN subset holds $Q_v = 2F_v + 1$ votes, it contains $n - f$ replicas.*
Proof: From Availability, it is always possible to access a quorum holding $Q_v = 2F_v + 1$ votes with at most $n - f$ replicas. In addition, the correspondent proof shows that such quorum contains all V_{min} replicas, which makes it a MIN subset. ■

Now we use Lemmas C4-C6 and Corollaries C3-C4 to prove the following:

Theorem C7: *All subsets holding $Q_v = 2F_v + 1$ votes are quorums that intersect in at least $f + 1$ replicas.*
Proof: From Lemma C6, we can infer that any possible quorum can be classified as either a MAX or MIN subset. We already showed that MAX (resp. MIN) subsets intersect in at least $f+1$ replicas in Lemma C4 (resp. Lemma C5). To finish proving this theorem, we need to show that, as long as they hold $Q_v = 2F_v + 1$ votes, any MAX and MIN intersect in at least $f + 1$ replicas. From Corollary C3, any MAX that is a minimal quorum (i.e., contains Q_v votes) contains $2f + 1$ replicas. From Corollary C4, any MIN holding Q_v votes must contain $n - f$ replicas. Therefore, if we sum the number of replicas contained in MAX and MIN, we have:

$$\begin{aligned}(2f + 1) + (n - f) = \\ 2f + 1 + 3f + \Delta + 1 - f = \\ 4f + \Delta + 2\end{aligned}$$

Furthermore, we can also observe that:

$$4f + \Delta + 2 \geq n =$$
$$4f + \Delta + 2 \geq 3f + \Delta + 1 =$$
$$2f + 2 \geq f + 1$$

Since $2f + 2 \geq f + 1$, MAX and MIN intersect in at least $f + 1$ replicas. Therefore, all subsets holding $Q_v = 2F_v + 1$ votes are quorums that intersect in at least $f + 1$ replicas. ■

We can finally prove the Consistency property using Theorem C7:

Theorem C8 (Consistency): *All quorums that hold Q_v votes intersect by at least one correct replica.*

Proof: Theorem C7 shows that gathering Q_v votes guarantee quorum intersection in $f + 1$ replicas. Since at least one out of those $f + 1$ replicas must be correct, if Q_v votes are gathered, we have quorums that intersect by at least one correct replica. ■