Lessons Learned with NavTech: a Framework for Reliable Large-Scale Applications

Paulo Veríssimo

DI–FCUL TR–02–17

December 2002

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at http://www.di.fc.ul.pt/tech-reports. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

Lessons Learned with NAVTECH: a Framework for Reliable Large-Scale Applications

Paulo Veríssimo
pjv@di.fc.ul.pt
University of Lisboa - FCUL*

Abstract

This paper presents an overview of the results of NavTech, an architectural framework to support the development of reliable large-scale applications. Several concrete protocols designed under the NavTech framework were published along the years, but most of the architectural innovations together with the skeleton protocols had remained unpublished. We identify the fundamental building blocks and the relevant connections among them that make it possible to support a wide range of applications. The architecture explicitly addresses the issues of scale by admitting clustering, and relying on a hierarchical network model (a WAN-of-LANs). Besides, it features a novel membership structure, based on the site-participant hierarchy. This membership skeleton supports in turn a hierarchy of protocol building blocks. Inside each block, different algorithms can be constructed by microprotocol composition, achieving different specializations of the architecture. NavTech protocols further enjoy the topology awareness property, which enhances their scalability.

1 Introduction

This paper reports the lessons and experience we gained in developing a framework called NAVTECH, aimed at supporting open large-scale distributed applications with emphasis on reliability and scalability. To put the subject in context, we claim that the fundamental problems that arise when building these applications are architectural rather than algorithmic. If the underlying architecture has the right constructs, useful properties emerge and help building efficient algorithms. Indeed we published a number of algorithms and protocols that took advantage of one such property, *topology-awareness*. In [39] a complete survey of the former can be found. However, most of the architectural innovations introduced by NAVTECH, together with its membership and failure detection *skeleton protocols*, had remained unpublished. The objective of this paper is to report these concepts, and the lessons we learned, in a comprehensive way.

When designing any form of support system for distributed applications, there is always a tradeoff between generality and performance. Generic approaches make few assumptions

*Faculdade de Ciências da Universidade de Lisboa. Bloco C5, Campo Grande, 1700 Lisboa - Portugal. Tel. +(351) 1 750 0087 (secretariat); +(351) 1 750 0103 (direct). Fax +(351) 1 750 0084. Navigators Home Page: http://www.navigators.di.fc.ul.pt.

about the underlying architecture and network. The resulting middleware and communication protocols are easy to port to different settings, but often exhibit poor performance and lack the ability to exploit advantages of the infrastructure. On the other hand, tailored solutions exploit particular features of given networks, or given host architectures, in order to achieve better performance. However, tailoring has its disadvantages: if the features exploited are peculiar only to one or two networks or host architectures, it may be difficult— or not rewarding— to port the resulting protocols to other settings that do not own these characteristics. The successful design must capture the right balance between generality and performance. If we focus our attention on the design of large-scale systems, this balance becomes a very delicate one. Firstly, because the opportunity for heterogeneity increases with the size and openess of the system. Secondly, because the large distances and numbers of participants involved welcome all efforts that can be made to gain in performance and scalability.

The NAVTECH *architectural constructs* aim at solving a few fundamental problems related with host and network architecture: the fact that network topology and hierarchy are most of the times concealed; the often neglected duality between processes and processors or, in our terminology, participants and sites. The functional aspects of the NAVTECH model are based on a few architectural paradigms, which address the scale issues just discussed and put in place a few hooks for topology awareness: 2-tier networking; clustering; site-participant multiplexing; scalable groups.

In short, NAVTECH foresees network infrastructures as composed of local clusters, interconnected by the long-haul links of a global counterpart. Furthermore, it assumes that both the local and the global parts have different interesting properties which protocols can take advantage from. As a matter of fact, this model can be applied to the majority of existing large-scale networks based on reliable high-speed local-area networks interconnected by slower long-haul connections (e.g. Internet, telecommunications). In the host scope, it assumes the separation between computing (the participant part), and communication (the site part). Finally, it assumes that the above-mentioned entities may form groups in either part (site or participant groups), and be managed as such (participant and/or site-group membership).

These constructs allow a designer using such a framework to draw conclusions that may be extremely relevant to the nature, performance and scalability of applications built on top of NAVTECH : the infrastructure is a *WAN-of-LANs*; protocols may extract advantages from being *topology-aware*; protocols for distributed computing support may be different from protocols for inter-site communication, this *separation of concerns* being advantageous for the simplicity of either one; *scalable groups* may provide the right construct to keep together significant numbers of sites or participants spread over an even more significant geographical area, on an infrastructure of uncertain connectivity.

Several models supporting different assumptions about the *correctness* of programs have been developed in this area (e.g. asynchronous, synchronous, or partially synchronous with failure detectors; view-synchronous; primary partition; partitionable, etc.). NAVTECH has attempted at defining a common denominator framework so that the infrastructure can be reused for the several possible models enumerated. In short, NAVTECH has a model of partial synchrony, which can vary anywhere in the space between synchronous and asyn-

chronous. Obviously, not all semantics are possible over the complete synchrony spectrum. *Skeleton protocols*, the only fixed algorithmic component of this framework, glue the several NAVTECH modules in a way that secures the desired semantics, by acting appropriately on the data flow path at precise points, called *taps*. The skeleton protocols, which are essentially related with failure detection and membership, are generic automata, with placeholders for functions that execute at certain points of system evolution.

In the classical debate pro/con groups, two issues have always emerged as being controversial: that groups are often tied to a given communication semantics (e.g. causal); that groups are heavyweight constructs and thus do not scale (e.g. beyond a few dozens). These criticisms are relevant, and we learned that from true experience in our experiments with groups. NAVTECH contributed to solving these problems by introducing a group model which stands on its own, independently from the communication semantics each instantiation may have. The groups are simple and composable for scalability. Different modules of a compound may have different semantics. The importance given to architecture and in particular the skeleton protocol approach— which gives structure to semantic independence and group composability— were but early symptoms of a syndrome of growing evidence in current research on group-oriented systems: importance is shifting from communication to membership and failure detection, and from sheer algorithmics to architectural composition of protocol modules made of microprotocols.

In summary, the NAVTECH framework specifies architectural constructs and mechanisms to assist in the design and execution of dependable distributed applications in large-scale settings. NAVTECH is essentially a macroscopic framework made of entities sitting on components such as the Internet, metropolitan area networks, private local networks, vanilla operating systems, satellite constellations, kept together by the glue of a run-time environment and a few protocols. The principles of NAVTECH are applicable to known infrastructural networking and computing technologies, and in that sense it is an open architecture, applicable to systems built from COTS components and sub-systems.

The next section, Section 2, reports related work. Then, the paper is organised as follows. Sections 3 and 4 discuss the system model and architecture. Sections 5 and 6 address the fundamental building blocks of NAVTECH , those concerned with failure detection and with membership. Section 7 presents the modules related with the data path, which host the Communication and Activity Services. Finally, the paper reviews several protocols that we developed on top of NAVTECH with the aim of validating our ideas, and concludes with a few retrospective remarks.

2 Related Work

Group-oriented systems are generic enough to support a mix of programming paradigms in the client-server and event-based, publish-subscribe areas. Over the years, they have proved to be an environment where it is straightforward to program complex reliable distributed applications. The latest generation systems have tried to solve issues such as large-scale, configurability and openess.

The NAVTECH work was not the first one to exploit topology information to improve

the design of group communication protocols. Several other examples can be found in the literature. However, with a few notable exceptions, most examples were focused on the solution of a particular problem, and have put their emphasis on algorithmics[4, 7, 8, 16, 23, 32]. To our knowledge, NAVTECH was pioneer in the coherent and systematic use of topology awareness, through the definition of a generic architectural construct, the WAN-of-LANs model. This enabled the application of topology awareness in a vertical manner, to practically all protocols developed for NAVTECH .

Totem [27] and Transis [3] are amongst the projects that have addressed the topology issue in a more systematic way. The Totem system provides total order multicast over interconnected local-area networks. The system is based on a composition of *intra*-LAN and *inter*-LAN protocols. The Transis system follows a network model that has similarities with the one followed in NAVTECH , and in essence their approach is complementary to ours: in Transis emphasis was given to efficient local-area communication and hierarchical composition while in NAVTECH emphasis was given on dynamic configuration and filtering of local control data.

Composition is an established principle of distributed protocol design. Typical examples of protocol composition frameworks are the x-kernel[29], Horus [40], Ensemble [22], Coyote [6]. The NAVTECH architecture is not tied to a concrete protocol composition framework, but it assumes that modules are implemented using such a construct. More recently, we are developing on the *Appia* framework[25]. One of the early prototypes of NAVTECH used the Horus run-time, combined with our own protocols[10]. A subset of NAVTECH protocols, namely the Light-Weight Group Service[36], have been implemented on the Ensemble system. Recent works address scalability through probabilistic algorithmic approaches, trading off determinism for scalability[21, 15].

3 System Model

Like in any other architectural framework, we paid special attention to the system model, and its main facets: network hierarchy; processes and processors; fault assumptions and synchronism. The functional aspects of the NAVTECH model are based on a few architectural paradigms, which address the scale issues just discussed and put in place a few hooks for topology awareness: 2-tier networking; clustering; site-participant multiplexing; groups as a scalable construct. The non-functional aspects of the model prefigure what we believe to be a reasonable assumption for a generic framework. In short, NAVTECH has a model of partial or uncertain synchrony, that can vary anywhere in the space between synchronous and asynchronous. The specific models that best reflect NAVTECH 's synchrony assumptions are the timed-asynchronous[12] and the quasi-synchronous[46] models, or the recent timely computing base model[41].

3.1 Topological Model

NAVTECH was designed for large-scale applications. Topology issues and communication system characteristics have a fundamental impact on the achievable scale of computations,

as systems grow in span, complexity and number of hosts. A large-scale network such as the Internet forms what we might call the *global network*, with a number of *structural* characteristics: sparse connectivity; limited diffusion capabilities; weak reliability and timeliness; globe-wide distances; public-domain or standard protocols. On the other hand, *local network* infrastructures take significantly different characteristics: availability of LAN or MAN technology; dense connectivity (normally broadcast-level); good reliability and timeliness; private operation, enterprise-oriented. This analysis identifies a fundamental topological paradigm retained by NAVTECH : its organization as a 2-tier WAN-of-LANs.

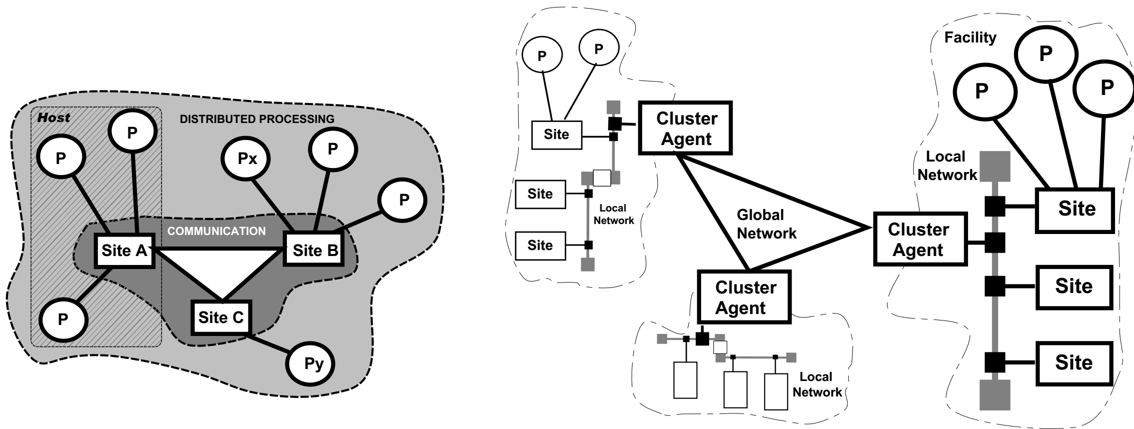

Figure 1: NAVTECH Topological Constructs: (a) Site-participant duality; (b) WAN-of-LANs

2-tier WAN-of-LANs

As we said before, large-scale computing infrastructures will retain a clear duality, which is materialized by several aspects, from administration to technology, in what appears to be a 2-tier infrastructure. Our model of WAN-of-LANs networks, consists of pools of sites with high connectivity links, such as LANS or MANs, or ATM fabrics, interconnected in the upper tier by a point-to-point global network. More concretely, we mean that: the global network is public and runs standard, de jure or de facto, protocols; each local network is run by a single, private, entity [1], and can thus run specific protocols alongside with or in complement to standard ones.

Site and Facility Clustering

A fundamental feature of the NAVTECH platform is to take into account the clustering naturally provided by the underlying network. In fact, clustering seems one of the most promising techniques to cope with large-scale, providing the means to implement effective divide-and-conquer strategies. In todays' networks, we identify at least two clustering entities: the *Site* as a cluster of participants; the *Facility* as a cluster of sites.

What we might see as a first level of clustering consists in the separation between communication and processing, by recognizing that the (sometimes large) number of

[1] E.g.: set of LANs of a university campus, MAN of a large industrial complex, LAN of a regional company department.

processes— the participants— that are engaged in distributed applications in a host, should explicitly share the communication resources used for networking— the site part.

The second clustering level is obviously compatible with the 2-tier architecture identified in the previous section: separation of communication amongst local aggregates of hosts, which we call facilities, from long-haul communication amongst facilities. Clustering of sites that coexist locally can simplify internetwork addressing, communication and administration. These sites are hidden behind a single entry-point, a *facility gateway*, a logical gateway that represents the local network members, for the global network. Organization-dependent clustering allows to run specific protocols behind the facility gateways, without that colliding with the need to use standard protocols in wide-area networking. Global network communication is thus performed essentially among facility gateways.

Sites and Participants

In NAVTECH , protocols implementing the interactions among entities in different hosts recognize the site-participant separation of concerns in the internal structure of system hosts, as represented in Figure 1a: the *Participant*, one from a collection of end entities performing distributed *activities* in different hosts (e.g. processes, tasks, etc.); the *Site*, the underlying entity clustering all participants residing in a same host, and performing *communication* on behalf of them (e.g. the host's communications server). The big picture of the NAVTECH architecture, showing the 2-tier WAN-of-LANs and clustering, is illustrated in Figure 1b.

Besides its practical relevance, the distinction between sites and participants in our model is also of major theoretical relevance. In fact, there are well-known results about the impossibility or difficulty of reliably detecting failures in distributed systems, depending on their asynchrony[18, 8]. A system recognizing the site-participant duality may perform failure detection more effectively: while site failure detection may be unreliable, participant failures can be reliably detected. This has the virtue of reducing the range of events that cause unreliable failure detection.

3.2 Group Model

The concept of group appears intuitively when describing many distributed actions[47]. In fact, grouping participants spread over a large-scale network seems to help solving a number of the problems underlying the correct execution of an application: connectivity, efficient addressing, reliable communication, consistency, and so forth. On the other hand, a current criticism of group paradigms says that group services do not scale well. This remark is particularly true when applied to systems not designed for large-scale, for example, offering just one model of group membership and communication protocols. It is impossible to provide the same type of performance and correctness guarantees, both to a group with thousands of members and to a group of a dozen members.

In NAVTECH , we address this contradiction through a composable group model. The basic group is simple, but complex and very large scale applications can be formed by composing several groups having different communication semantics. These groups can also be

mapped both onto the interaction style and onto the topological structure. For example: a group of a thousand members for dissemination with lightweight communication, versus a group with a dozen members for managing a server replica set with heavyweight communication; or a hierarchy of groups whose top level exists at the Global Network level, with its members representing lower-level groups inside Local Network clusters.

The 2-tier Unidirectional Group Model

One important characteristic of our architecture is the 2-tier separation between:

- **groups** of participants as users of the platform, in the sense normally used in other works, i.e. sets of application processes;
- **site-groups**, as the set of sites where the entities related with a participant group reside.

Groups are concerned with the consistency of distributed activities, whereas site-groups are concerned with the correctness of communication. For example, in Figure 1a, the site-group corresponding to group $\langle P_x, P_y \rangle$ is $\langle B, C \rangle$. In consequence of this separation of concerns, the membership and failure detection of participants and sites are decoupled. Participant membership and site membership are concerned with groups and site-groups, respectively. The corresponding guarantees about failure detection, are reliable in the former, unreliable in the latter. This model is at the heart of the two-tier membership structure of NAVTECH , detailed in Section 6.

The next important characteristic of the NAVTECH group model is the drastic difference in the role of senders and recipients of messages. The model distinguishes between:

- **members** of the group, who are *only* recipients of messages;
- **senders**, who *only* send into the group.

It is thus a **unidirectional** group model, depicted on the left of Figure 2, which presents the "open-loop" view of the information path from senders to members (recipients). However, the classical bidirectional access to a group is also easily achieved, as depicted on the right of Figure 2. Sender and member endpoints are denoted S and M, respectively, in Figure 2 and the ones that follow. Participants, depending on whether they are going to send, or receive, or both, acquire a sender or a member interface to the group, or both. Besides, a participant can be sender to, or member of, several groups. At this point, we can also be precise about the meaning of group and site-group. The members of a group are *exclusively* the participants who receive messages sent to the group. In the example on the left of Figure 2, the group G_i has two members that reside in sites A and B. On the other hand, a site-group is in charge of performing inter-site communication correctly and efficiently. In consequence, there will be a site-group member in every site where there is activity related with the corresponding group. In the example, there are senders to the group in sites B and C. This means that considering both the members and the senders involved, the corresponding site-group SG_i includes sites A, B and C. In conclusion, a *site-group* has the following characteristics:

- it is formed by the sites (one element per site) that hold at least one sender or member of a group;

- it implements group addressing by mapping group names on site addresses (network hosts);
- its membership includes the delivery set of reliable protocols (i.e. determines the acknowledgments to be expected);
- it implements the protocols that perform communication *from within* that group;
- it is a contact point for protocols that perform communication *from outside* that group.

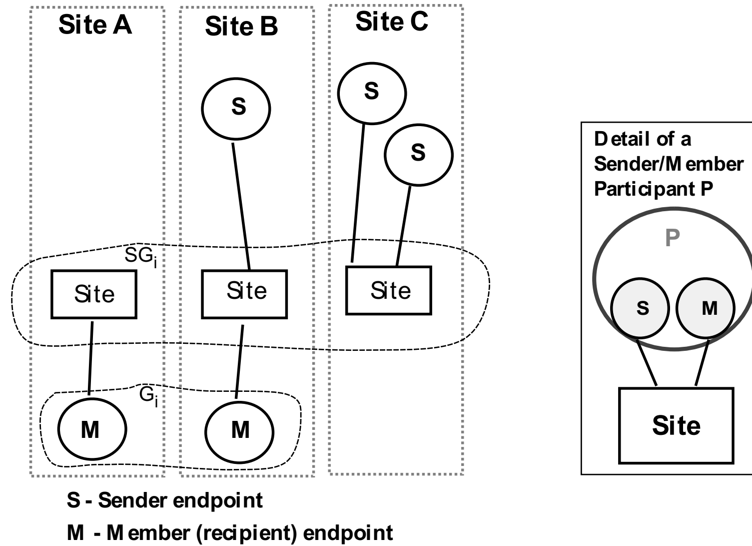

Figure 2: NAVTECH 2-tier unidirectional group model

Thinking of the system as having two levels of abstraction, site- and participant-level, may significantly improve its performance and scalability[47]. A site multiplexes all the local participants related with a given group, through a single communication endpoint (the local site-group member) that represents the group for the network. Site-group members multiplex local senders-to and members-of, a given group. The practical implications are that if there are more than one member of a group in a site, only one message is sent there, and then copied to all recipients. Likewise, when a site fails, a single run of a site failure detection algorithm needs to be executed, instead of having many runs executing in parallel as in some other earlier generation groups systems[33, 26, 7]. On the other hand, site-group members process the send requests from their local senders, and run the protocols that ensure delivery to the members, with the requested guarantees. Senders *are not* full-right members of a group. They get from the system just the necessary information and support to `send` to the group with the required quality of service, but without further overhead. The first participant *joining* a group creates the corresponding local site-group member. Only members partake whatever functions concern the management of the group, and in consequence pay the associated cost. This separation of roles is paramount to our claim of obtaining a simple but scalable group model, by composition. Consider a traditional heavyweight and bidirectional group model, that is, one where the separation site-participant does not exist, and where each endpoint has to cope with both transmission and reception: it is quite awkward to build more complex applications by using that kind of groups. As we show in the next section, building applications by composition with our group model is conceptually very

simple.

Composing Groups

A 2-tier unidirectional group model is a very simple building block for more complex applications. Real program entities (processes, tasks, threads, etc.) will incarnate senders and/or members on a need basis, and thus will only be as complex as required.

For example, a producers-consumers application will fit directly onto unidirectional communication, by providing the producers with simple send access to the group of consumers, and giving each consumer a member status. Likewise, a multipeer conversation is configured by having each entity become both a sender and member participant for that group.

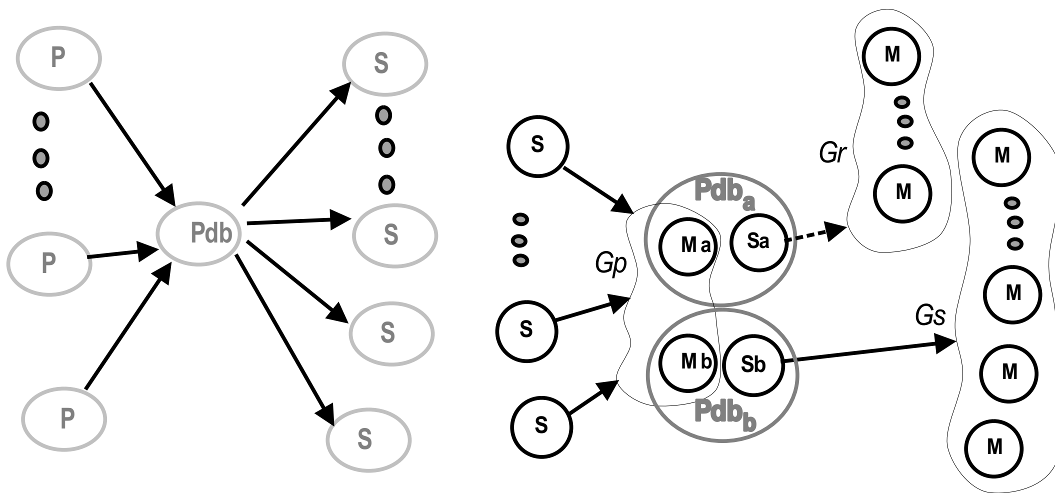

Figure 3: "Publishers/subscribers" application

The right of Figure 3 shows how to implement the well-known publishers/subscribers paradigm, depicted on the left. P_{db} is the publishing server implementing the message bus, which is made reliable by replication. Publishers simply acquire sender interfaces to the group of replicas G_p. This is combined with dissemination to the subscribers groups, which may be more than one (G_r,G_s), according to the subscription semantics and interests. This task can either be centralized in a main replica, or divided by the replicas, as shown in the figure: the server replica $P_{db}(a)$, while receiving from its member interface M_a to group G_p, disseminates chosen information to the group of subscribers G_r, through its send-only interface S_a to the group.

An interesting feature of our model is revealed in this case: the independence of communication and activity semantics from the group model, and its effect on scale. We know from experience that some algorithms do not scale. The question is whether all algorithms have to scale. By analyzing the semantic needs of the main interaction styles (client-server, multipeer, dissemination) we arrived at the interesting conclusion that scale is bound to semantics of communication. In short, the greater the scale of computations, the lesser the semantic guarantees required of communication, and protocols may take advantage of that.

One approach pursued in NAVTECH is, for example, preventing semantically "heavy-weight" communication from occurring outside the local network scope. In the example

above, the client-server publishing database may be accessed through atomic multicast communication and primary-partition replication management. However, dissemination to subscribers can be implemented through a weaker QoS, for example best-effort multicast, to comply with the different scale needs of this part of the interaction. Scale requirements are fulfilled because of the freedom of the designer to select the adequate semantics for each group. These conjectures were validated by several of the algorithms and protocols developed (*see* [39]). Recent protocols exploit semantics in an even more sophisticated way[28].

3.3 Fault and Synchrony Model

Formalizing, we assume a system model where a set P of participants noted $P_i, P_j, ...$ exchange messages relying on the respective sites $S_p, S_q, ...$. Sites are interconnected by a communication network and execute communication protocols on behalf of the participants. Communication and activity are group based, where if a set of participants forms a *group* G_i, then the set of sites involved in communication to and within that group forms a *site-group* SG_i.

We assume the system to follow an omissive failure model, that is, components *only do* timing, omission and crash failures, and no value failures occur. More precisely, they only do *late* timing failures. In what concerns processing or communication delays, the system can have any degree of synchronism, that is, if bounds exist, their magnitude may be uncertain or not known. The only strong assumption about time is that local clocks exist and have a bounded rate of drift towards real time.

What can or not be done in a distributed system depends on how faithfully a crash failure can be detected[8]. Normally, this amounts to being able to specify what is a detectable omissive failure[44]. Crash failures are particular cases of timing failures, where a process produces infinitely many timing failures with infinite lateness degree. In consequence, an omission degree[42, 44] specification is the most usual way of specifying a detectable crash: timing failures beyond a certain lateness degree (amount of delay) are transformed into omissions through a timeout based detector. Then, going beyond an omission degree threshold (number of successive failures) is finally considered a crash.

Different instances of the NAVTECH framework, and/or specific protocol combinations, may depend on additional synchrony and fault assumptions. For example, if the system synchrony is such that a perfect failure detector cannot be built, then many practical problems have no deterministic solution. However, if an eventual strong failure detector can be built, then problems such as atomic multicast delivery are possible under certain conditions[8].

Recall that NAVTECH is a framework. In consequence, the contribution of this work is to provide the best possible architectural constructs to improve the implementation of large-scale distributed systems, in complement to and in assistance of algorithmic approaches enhancing scalability. The latter have been exploited in the several algorithms and protocols we published over the past few years[39], whereas the main focus of the rest of the paper lies with the former.

4 The Architecture of NAVTECH

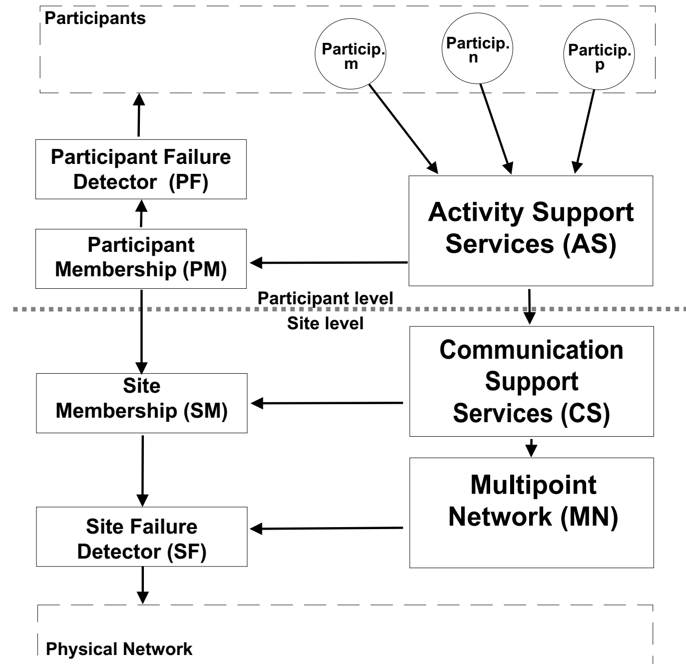

Figure 4: NAVTECH Architecture: 'depends-on' relation

The overall architecture of NAVTECH is represented in Figure 4, illustrating the 'depends-on' relation between modules. The NAVTECH platform lies on an abstraction of a multipoint network, the *Multipoint Network, MN*, created over the physical infrastructure. Its main properties are the provision of multipoint addressing and a moderate partition healing ability, both depending on topology information. The *Site Failure Detector (SF)* module is in charge of assessing the connectivity to and liveness of remote sites, and depends on the Multipoint Network, by listening to traffic going into each site, and by exchanging information with other SF modules. The *Site Membership (SM)* module depends on information given by the SF module. Based on the latter, and on interface calls, it creates and modifies the membership and view of site-groups. The *Communication Support Services* module, *CS*, implements reliable communication and clock synchronization. It depends on information given by the SM module, about the composition of recipient groups, for example to determine where acknowledgments are expected from, to inform users about changes, or to ensure that a message is delivered to a given group composition. It implements protocols and algorithms that allow participants to communicate between sites, from best-effort to reliable or atomic multicast.

Together, these modules form the 'site' part of a host. All the four modules described are topology sensitive, that is, protocols may run differently depending on the following conditions: 'this host' is in a local network; 'this host' is in the global network; 'this host' is a Facility Gateway.

The *Participant Failure Detector* module, *PF*, is a module with strictly local operation: based on probes implemented with O.S. support, it assesses liveness of participants, accordingly to pre-established failure criteria. Note the decoupling between site and participant failure detection whose virtues have been discussed earlier in this paper. The *Participant*

Membership module, *PM*, depends on information propagated by the SM and the PF modules. Based on the latter, and on interface calls, it creates and modifies the membership of participant groups, and validates activity conditions, such as, for example, *majority* in a primary partition. The *Activity Support Services* module, *AS*, depends on the group communication and clock services provided by the CS module, and on the membership information provided by the PM module. It implements protocols and algorithms that assist participant activity, such as replication management, mutual exclusion and concurrency control, cooperation awareness, etc. These three modules form the 'participant' part of the host.

Although not shown for simplicity, there can be several of these participant-level modules mapped onto the site part, for example representing different unrelated application support environments which make use of the common communication services. Likewise, in the same AS module there can be several groups of participants mapped onto one site group, in what is called a *lightweight group* structure. NAVTECH uses groups as its central paradigm, but has a separation of concerns that is beneficial to system composability and scalability: (a) a participant group G_m is concerned with the correctness and consistency of the *execution* of distributed applications, and relies on the assistance of the algorithms in the AS module for that purpose; (b) G_m members delegate *communication* on the site group SG_m, which runs communication protocols served by CS.

The membership service is the spinal cord of NAVTECH . As depicted on the left of Figure 6, there is a chain of information between membership related modules, down from the site Failure Detector module up to the participant Failure Detector. This information is used to act on the group communication and activity data flows, ensuring that the correctness conditions of the chosen semantics are met. The membership structure intersects the data flow through constructs called *taps*, shown in the figure, at the bottom of the Communication Support (CS) and Activity Support (AS) modules. The *taps* perform filtering, multiplexing and demultiplexing functions, which will be detailed later in the text.

By acting on the data flow through the several taps, the membership/failure detection protocols enforce typical baseline semantics, defined in terms of *profiles*, which we detail below and depict in Figure 5.

The Multicasting tap enforces a weak-partial membership, depending on the failure information supplied by the SF module, which keeps track of the hosts that are mutually connected. This way, it secures the *best-effort* profile, where views are mainly supposed to assist multicast addressing. Several views may thus be formed in a system, namely in the presence of partitions. Those views reflect in the best manner possible the set of sites that are mutually reachable, but no further guarantees are given. The best-effort profile can use protocols further up, but it bypasses all filters up to the participant, as sketched in Figure 5, that is, the other taps are so to speak forced open at all times.

The Comunication tap establishes a hook between site membership and reliable multicast protocols, to ensure they have the property characterizing the *view-synchronous* profile, namely, a strong-partial membership, meaning that several views may exist, but their intersection is empty. Informally speaking, this means that the tap opens and closes under command of the site membership protocol, and may even divert the data flow through the Site Membership module, in order to enforce the desired semantics. This profile allows a framework for building applications of the kind explored for example in [4]. The

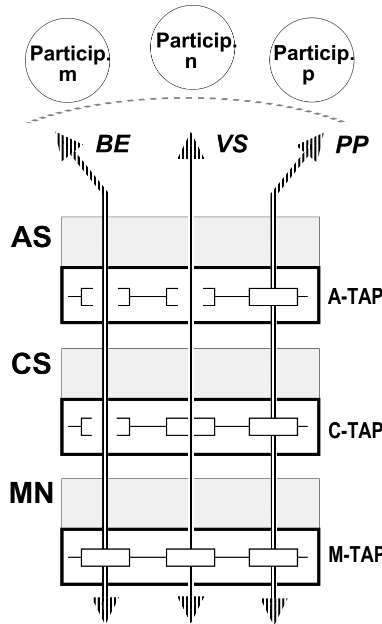

Figure 5: Data flow through the NAVTECH profiles

view-synchronous profile can use protocols further up, but it bypasses all filters up to the participant.

The Activity tap establishes a hook between participant membership and activity management protocols (e.g. replication management), to ensure they have the property characterizing the *primary-partition* profile, namely a linear membership, based on the majority criterion. This means that a single view is allowed to be in activity, at all times. This tap operates in a way similar to the Communication tap.

Note that NAVTECH allows the definition of other semantics besides the typical ones, implemented by disabling (opening) any of the taps and constructing ad-hoc protocols in the corresponding module. For example, optimistic consistency approaches might be achieved by: using CS protocols on top of the view-synchronous profile; bypassing the (primary-partition) Activity tap; and implementing AS protocols to achieve the desired semantics. The well-known works based on failure detectors to implement consensus and atomic multicast to fixed destination sets (static membership) can be implemented bypassing the Communication tap and building the relevant protocols in the CS module.

5 Failure Detection

A NAVTECH Domain is the collection of machines where a collection of related NAVTECH software is supposed to run. Given that a NAVTECH Domain is foreseen to run on global, world-wide network infrastructures (e.g. Internet), there is a registration procedure, not detailed here, whereby new sites (machines) get to be known to the other machines in the Domain. From now on, when we say 'all sites' we mean all sites registered in a NAVTECH Domain.

Given the uncertain degree of synchrony assumed for NAVTECH , site failure detection may be *unreliable*. However, our concern in this paper is to define a failure detection struc-

ture and explain the correctness of the relevant mechanisms, independently from the fault and synchrony assumptions of the system. The structure is modular, and the only thing that changes with changing assumptions is the particular implementation of some of the modules.

5.1 Site Failure Detection

The Site Failure Detection Service is performed by the module with the same name, SF. The objective of the service is to assess the state of the sites running distributed applications under NAVTECH .

The majority of detectors known are *crash* failure detectors. Its utility has been discussed in several works both in the asynchronous and synchronous worlds. A known paper by Chandra & Toueg has established a hierarchy of crash failure detectors, and the problems they can solve[8]. The most powerful is a *perfect crash failure detector*. Then, accordingly to decreasing faithfulness of the detector, in the measure that synchronism decreases, we have less and less accurate detectors, with weaker properties.

It has been shown by Chandra & Toueg[8] that detectors need *accuracy* and *completeness* to be effective. More recently, it has been shown[17, 2] that if they could further detect timing failures in a timely manner, the system would be able to address timeliness problems, even if being almost asynchronous or imperfectly synchronous. Timed *accuracy* and *completeness* were defined in [41], allowing to characterize timing failure detectors.

Usually, the output of failure detectors serves the purpose of system reconfiguration, for example, a progress condition of a protocol, a new view in the membership of a group. However, when: (a) the criterion for a detector to declare crash of a process or site is static; (b) the semantics of such detection is poor, then chances are that the system: (a') will not be able to adapt to the changing environment; (b') will suffer from the many mistakes of the detector.

Triggering reconfiguration using fixed and/or aggressive timeouts may cause instability problems in settings of uncertain timeliness, such as large-scale networks. The effect on replicated, collaborative, or QoS-sensitive applications would be disastrous, with the system configuration bouncing back and forth at an unbearable rate. We can imagine participants doing nothing but coming in and out of collaboration groups, servers disapearing from a replica group or joining and leaving without rest, multimedia streams going back and forth from colour to B&W, or high to low compression.

The detectors proposed by Chandra try to overcome the uncertainty of the environment's synchrony (is a site slow or crashed?) by successively increasing the timeout period. But the detector is still looking for a crash. We have introduced *QoS failure detectors*, which finesse the semantics of failure. Observe that omissive failures are a spectrum that goes from timing, through omission, to crash. Omission failures are timing failures with infinite lateness degree. Crash failures are infinitely many omission failures.

We define the symptom of QoS failure using the multidimensional quantitative measures one normally includes in the notion of quality of service (QoS) or level of service (LoS)[11], such as bandwidth, latency, omission degree or rate. The semantics of our de-

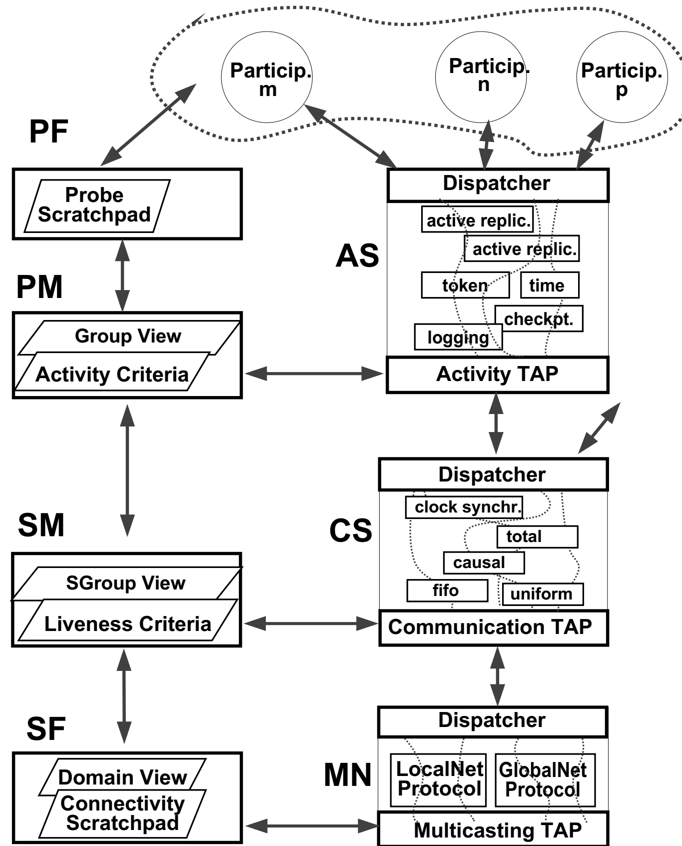

Figure 6: Detailed View of the NAVTECH Modules: 2-tier membership and failure detection structure (SF, SM, PM, PF); data flow modules (MN, CS, AS)

tectors becomes richer, and hence they do less mistakes.

5.2 Failure, Suspicion and Detection

Each site has a local site failure detection module SF. The main data structures of a local SF, as shown in Figure 6, are the *Connectivity Scratchpad*, C_{sp}, and the *Domain View*, $V(D)$. Given a Domain $\mathcal{D} = \{S_1, S_2, ..., S_n\}$, the C_{sp} holds several kinds of information: the local SF's opinion on each site; the other SFs' opinions on each site; the result of failure detection tests. $V(D)$ contains the current global opinion about the hosts that are alive. A failure detector protocol running among all SFs maintains $V(D)$ up to date, by processing the information contained in the scratchpads and getting to a consensual decision on it. For the sake of scalability, each site only needs to keep track of sites registered in the Domain that are currently members of a site-group which the local site belongs to.

For the purpose of detection, a site has a failure if it does not comply with a specification $Spec$, which is a proposition of behavior in the omissive domain. For example, a site would be crash-failed if it did more than k successive omissions (e.g., failing more than k 'I'm-alives' or 'heartbeats' in a period of reference); or, it would be QoS-failed if it exhibited an error rate higher than e, or a roundtrip time higher than d. Besides distinguishing between crash and QoS failure, we also distinguish between local suspicion, under the responsibility of a single module, and global detection, based on the opinion of all detector

modules. Finally, unlike all failure detector works we are aware of, we allow different sets of sites to behave differently (i.e., follow different $Spec$s) according to the application needs they serve. Consider $Spec$ containing the definitions of correct site behavior for some set of sites $\mathcal{S}$ ($\mathcal{S} \subseteq \mathcal{D}$). Then:

Local Suspicion Vector (LSV_u) - a vector of Booleans with $\#\{\mathcal{S}\}$ positions, for each local site $S_u \in \mathcal{S}$, such that $LSV_u[v]$ contains the opinion of S_u about every other $S_{v \neq u}$; additionally, position $LSV_u[v] = 1$ iff the behavior of S_v as seen from S_u exceeded (was worse than) the thresholds imposed by $Spec$; it contains 0 otherwise.

Crash Failure Suspicion

Consider $Spec$ to be a specification of the crash failure class, such as used by canonical failure detectors (e.g. 'heartbeats'). In consequence, we define *crash failure suspicion*:

A site S_u suspects site S_f is crash-failed iff: $LSV_u[f] = 1$, for $Spec \equiv$ crash failure

At this point, the reader will note that each local SF instance has a Local Suspicion Vector, LSV. Now, we discuss how to transform our crash detector onto a semantically richer quality-of-service failure detector.

QoS Failure Suspicion

Instead of the relatively poor crash semantics, the failure detector can be configured to detect Quality-of-Service (QoS) failures, if we redefine $Spec$ to be in fact a *QoS specification*. This does not reduce the generality of our work, since a crash failure is a subset of a QoS failure, where the expected QoS is for the site to be up. The main difference is that so to speak we have introduced *analog* (versus binary) *failure detection*. We summarize the QoS failure detection mechanism below, the complete treatment can be found in [11, 43]. On the other hand, the QoS failure detection concept is a very broad one, and can reach parameters such as disk and network bandwidth, processor usage, video quality, etc., for processor resources as well as communication. A good example is QoS measurement among multiple multimedia rendering processes in a site. The following definitions are extensive to that realm.

A QoS $Spec$ is based on a set of parameters, referring to time-domain parameters, such as roundtrip delay, throughput and omission error rate. For each parameter P, we define: the sampling period, or interval of reference; the threshold (TH) not to be exceeded (positively or negatively) by the value of the parameter during that period; and the weight (WT) or relative importance of the parameter in the final result. Each parameter of the QoS specification is evaluated by the QoS-FD at each site S_u, for every other site $S_{v \neq u}$ of the set. We note the value V associated with parameter P kept at local site u about some remote site v as $V_u^v(P)$. For instance, $V_u^v(roundtrip)$ denotes the roundtrip delay between sites S_u and S_v as measured at S_u.

Once a threshold set, what is recorded over time by S_u is the percentage of sampling periods where it it was exceeded, for each S_v as seen from S_u. We call this variable *Threshold Exceeded (TE)*. For example, $TE = 2$ for round-trip time means that in 2% of the times the round-trip time threshold was exceeded. In other words, over time, TE gives a measure of

the current coverage of the threshold assumption for a given P. In this last example, the round-trip time assumption is holding with a probability of 98%. The weighted average of the TE of all parameters P, for each S_v as seen from S_u, gives the Disturbance Index:

$$\mathrm{DI}_u^v = \sum_P \mathrm{WT}(P) * \mathrm{TE}_u^v(P) / \sum_P \mathrm{WT}(P)$$

A value of DI_u^v exceeding a DI *threshold* means a QoS failure of S_v, in the opinion of S_u. This is reflected in the LSV evaluation, by setting $LSV_u[v] = 1$. DI is computed periodically for each S_u in $\mathcal{D}$, as specified by the QoS sampling period. After the computation of the DI, TE variables are reset, and incremented from zero until the next computation of DI. A very low DI value means that the QoS in the current set is within satisfactory thresholds.

Consider $Spec$ to be a QoS failure class specification. In consequence, we define *QoS failure suspicion*:

A site S_u suspects site S_f is Qos-failed iff: $LSV_u[f] = 1$, for $Spec \equiv$ QoS failure

This means that we perform essentially analog failure detection, and only at this point we do the analog-to-digital conversion of the local failure symptom. The reader will note that the QoS failure detector is the most general kind of FD, crash failure detection being a special case of the former.

General Failure Detection

We now discuss the issue of general failure detection in our system (be it resulting from crash or QoS suspicions). Recall that each local SF instance has a Local Suspicion Vector, LSV. Now it is necessary to test each other's opinions, to assess the symmetry and transitivity of the site's connections. Failure detector modules should actively cooperate to reach a decision as closer as possible to the true state of sites, and as unanimous as possible, in fact, to reach a decision about *failure detection*, from these suspicions.

It is known that these attributes, named accuracy and completeness[8], assume a range that depends on the system fault and synchrony model. In NAVTECH we put the adequate hooks in place for any kind of detector to be built (without loss of generality, recall that $\mathcal{S} \subseteq \mathcal{D}$):

Global Suspicion Matrix (GSM_u) - a matrix of Booleans with $\#\{\mathcal{S}\}^2$ positions, formed by the LSV of all sites in $\mathcal{S}$ as received by each local site S_u, such that each row $GSM[u, -]$ is LSV_u. If site S_u suspects S_v then $GSM[u, v]$ is true, and false otherwise. If all sites suspect S_v, then column $GSM[-, v]$ is true. We consider the diagonal $GSM[x, x]$, $\forall x$, as don't cares

Decision Function- the specification of a convergence function over GSM, whose result, each time it converges, yields the new state of $\mathcal{S}$, possibly implying the failure detection of one or more sites

Global State Vector (GSV_u) - a vector of Booleans with $\#\{\mathcal{S}\}$ positions, as output from the convergence function for each local site S_u that has reached a decision, such that

$GSV_u[v] = 1$ only if S_v was *detected* failed (rather than just suspected); it contains 0 otherwise

Local SF's exchange suspicions (LSV) each time they occur, and accumulate these suspicions in a Scratchpad, C_{sp}. At relevant points of the FD execution, the convergence function is run on the GSM, to produce a failure detection decision. The decision may concern more than one site simultaneously.

Consider a "Chandra/Toueg" strong failure detector. For such a detector to be built, the collection of decisions at all sites would have to be such that the resulting matrix of GSV's should yield: for every failed S_v, $GSV[u,v] = 1$ for all S_u; and there exists at least one S_w such that $GSV[u,w] = 0$ for all S_u. A convergence function detecting that property over all SF modules would converge upon any failure. On the other hand, a convergence function for an eventual strong failure detector, would converge upon a finite but unknown time.

There are other possibilities though. A refinement we introduced in failure detection in NAVTECH is the possibility of placing more trustworthiness on the decision of some local FDs than on others'. This is used to tune algorithms based on topology awareness or participant role. Consider the distinguished role of the facility agent, for example, as the mediator between the local and the global network. Likewise, in cooperative applications, the 'social role' is paramount: for example, in a tutorial over the network, the opinion of the lecturer's connectivity to the participants is more important than each others' connectivity[10]. This enhancement is detailed in [11, 43].

In conclusion, consider $Decision$ to be an adequate function to detect the failure of a site, such as would be used by canonical crash-only detectors (e.g. 'strong failure detection'). In consequence, we define (general) *site failure detection*:

A site S_v detects site S_f as failed iff: $GSV_v[f] = 1$ after a decision function converged

No matter what kind of decision about failure detection, the SF protocol should notify the relevant modules, namely the Site Membership module. Nevertheless, it would be good if this information were postponed, in situations of high but short-lived network instability, in order to avoid undesirable bouncing between costly system reconfigurations. We describe mechanisms for improving failure suspicions next.

Recovery

So far, we have left recovery out of this discussion, not to distract the reader. We introduce it now: our detectors are symmetric, and might be called *failure/recovery* detectors. In a generic framework, detectors must encompass the fact that in some models, processes do recover[31, 14, 1]. In the definitions we have made, recovery may be substituted for failure. For example, given a previously failed site S_f, a site S_u *suspects* site S_f is *alive* iff $LSV_u[f] = 0$. Likewise, a site S_u *detects* site S_f as *alive* iff $GSV_v[f] = 0$ after a decision function converged.

5.3 Improving Failure Detection Accuracy

In this section we discuss mechanisms which, albeit improvements to our generic failure detection framework, go beyond mere engineering optimizations. We propose a low-pass filter or *debouncer*, to integrate suspicions, and a *partition healing* facility. The purpose of both is to delay failure detection notification and allow for the system to stabilize. In both cases, this is done when there is sufficient evidence that we are in face on a short-lived instability period. Notice that none of these overrides known theoretical impossibility results. Our experiments have however shown that these measures drastically improve the behavior of FDs in systems of uncertain timeliness.

Suspicion Debouncing

Let us consider the failure detector performs series of 'test epochs', and the LSV is evaluated after each test epoch. We redefine suspicion, by introducing two states of suspicion: a site is suspected faulty if it fails a test, and is suspected failed if it fails k_f successive tests. In consequence, we define *fault* and *failure* suspicion:

A site S_v suspects site S_f is faulty iff: $LSV_v[f] = 1$ after a test epoch

A site S_v suspects site S_f is failed iff: $LSV_v[f] = 1$ for k_f successive test epochs

Conversely, a site is suspected to be *recovering* after the first epoch where $LSV_v[f] = 0$, but is not considered *alive* until being recovering for k_r successive test epochs. We allow the system to give different values to k_f and k_r just for versatility reasons. The decision function is not activated until a site is suspected 'failed' or 'alive', but the 'faulty' and 'recovering' suspicions are disseminated as *gossips*, namely to improve accuracy of the partition healing mechanism that we discuss next.

The effect of this mechanism is akin to combining two hardware mechanisms: the mechanical or electronic debouncers of digital switches, and the noise supressors of transmission lines. When a switch is closed, contacts physically bounce, such that the logical value switches several times between '0' to '1', before assuming the final value. A system inadvertently designed will register an enormous number of transitions, instead of only one, as would do a debounced switch. On the other hand, noise in a digital line whose steady state is '1' may sometimes cause it to bounce to '0' for short times. A noise supressor will filter out any such bounce whose duration is less than a specified 'low' pulse. Achieving the equivalent of these two mechanisms is our objective.

Partition Healing

Partitioning occurs when a network is split such that sites cannot continue to enjoy the symmetry, transitivity and connectivity properties. Telling network partitioning from site failure is useful, because the system support can do things in order to repair the problem, or avoid a premature declaration of failure of the out-partition sites:

- the simplest is for the SF to wait, during a given timeout, for a possible merge, instead of immediately declaring failure of the affected sites. This has the virtue of preventing the application from reconfiguring twice (back and forth) when the partition occurs for a short period of time.
- more sophisticated, but quite effective, is partition healing. Besides preventing premature failure declaration as well, it also attempts at removing the physical symptoms of partition.

  For a site S_v, a site S_p is partitioned if at least one of the following predicates hold:

Non-transitive Partition - there is a site S_m, such that: S_v is alive for S_m, and S_p is alive for S_m, and S_p is failed for S_v

Non-symmetrical Partition - S_p is failed for S_v, and S_v is alive for S_p

Total partition - there is a partition P_{out}, to which S_p belongs, and there is a partition P_{in}, to which S_v belongs, such that: all sites in P_{out} are failed for all sites in P_{in}, and all sites in P_{in} are alive for all sites in P_{in}

In the non-transitive partition example it is easy to imagine the decisions produced by successive test epochs, if influenced in turn by S_v, S_m, and S_p: $\langle l, m \rangle$, $\langle l, m, p \rangle$, $\langle m, p \rangle$. If propagated up, this will cause *membership instability*, obviously undesirable. Addressing this problem on a best-effort basis inside NAVTECH is not only possible but straightforward to implement. Whilst IPv6 is bound to improve routing responsiveness, connectivity glitches still last longer than desirable, for example due to the load queues that build up[24]. The above notion of partition can be extended to several sets of partitions. When the SF has additional information on the state of the network, provided by the MN for example, it may be possible to correlate failure of GlobalNet links to the defaulting route, in case of partial partition, or to the composition of P_{out}, in the case of total partition.

5.4 Failure Detector Skeleton Protocol

In this section, we present the skeleton protocol for the failure detector. Recall that the NAVTECH framework leaves room for several failure detection semantics. This is achieved by defining the skeleton in terms of macro functions that can be implemented in different ways. Several known failure detector algorithms can be embedded into the skeleton protocol and take advantage of the added functionality provided by NAVTECH , such as QoS failure detection, suspicion debouncing and partition healing.

Basic Failure Detector

The basic failure detector is based on having a test function run forever, $testEpoch$ in line 11, with a specification $Spec$, a target set of sites $Target$, which may be the domain or one of the site-groups, and a period $SamplingPeriod$.

```
// LSV: (1)-faulty; 1- failed; (0)- recovering; 0- alive

10 do forever
11     LSV_u = testEpoch(Spec,Target,SamplingPeriod)
12 od

15 when changed(LSV_u) do
16     broadcast(LSV_u)
17     decide(GSM_u)
18 od

20 when received(LSV_v) do
21     update(GSM_u)
22 od

25 when GSV_u=decided(GSM_u) do
26     notify(GSV_u)
27 od

30 when LSV_u[v] = faulty during k_f consecutive epochs do
31     LSV_u[v] = failed
32     broadcast(LSV_u)
33     decide(GSM_u)
34 od

40 when LSV_u[v] = recovering during k_r consecutive epochs do
41     LSV_u[v] = alive
42     broadcast(LSV_u[v])
43 od

45 when partitioned(GSM_u) do
46     reRoute(GSM_u) during k_h epochs
47 od

50 when healed(GSM_u) do
51     restoreRoute(GSM_u)
52 od
```

Figure 7: Failure Detector Skeleton Protocol (For each S_u)

Whenever, in result of the tests, the state variable changed(LSV_u) is true, meaning that the Local Suspicion Vector has changed, that information is broadcast to the domain (lines 15-16), and the sites immediately update their Global Suspicion Matrices (line 21). Immediately after, a decision process is invoked among the sites taking part in the detection process (line 17).

Recall that the decision function may assume different forms, that will dictate the failure detector semantics, and also the probability of convergence, but the skeleton protocol does not change. In Section 5.2 we have exemplified this point inspired by some of the asynchronous failure detector semantics presented in [8].

QoS Failure Detection

Let us go back to the test epoch. Introducing QoS failure detection does not modify the generic protocol, just the implementation of the $testEpoch$ function. The QoS test semantics is defined by external requests, through variable types exported by the SF service, their values being set in a user-dependent fashion. The test is mainly implemented by: (a) eavesdropping on source addresses of incoming messages; (b) sending explicit probe messages to complement the former (sites from where nothing arrives for too long); (c) periodic messages exchanged by SF modules.

The detector we designed for NAVTECH is oriented to assess connectivity, and thus measures: roundtrip delay, throughput and omission error rate. A QoS specification oriented to connectivity and an example test epoch are illustrated in Table 1.

QoS Specification				**Test Results**										
				host$_1$		**host$_2$**		...	**host$_u$**		...	**host$_n$**		
Parameter	**TS**	**TH**	**WT**	**TE**	V_u^1	**TE**	V_u^2	...	**TE**	V_u^i	...	**TE**	V_u^n
roundtrip (ms)	0	100^-	2	5	120	30	250	...	-	-	...	0	30
throughput (Kb/s)	100m	50^+	1	0	90	20	5	...	-	-	...	0	70
omission rate (/s)	10m	1^-	1	0	0	0	0	...	-	-	...	0	0
...	.	.	.	.	.	.	.	...	-	-	...	.	.
Disturbance Index (DI)	10s	5^-	-	2.5		20		...	-		...	0	
				↓		↓			↓			↓	
Local Suspicion Vector LCV$_i$ →				**0**		**1**		**...**	**-**		**...**	**0**	

Table 1: QoS Specification for a Test Epoch and Example Results at Site S_u

Instability Dampening

Failure detection can be dampened in the interest of system stability, by introducing suspicion debouncing. Recall that this is done by introducing two states of suspicion. The test epoch is modified to change LSV positions to `faulty` after a single run. Furthermore, the state variable *changed(LSV_u)* only admits `failed` and `alive` values, not `faulty`. As shown in lines 30-31, with debouncing it takes k_f test epochs for a site to be suspected failed. Since the action in lines 15-18 is only performed when and LSV position is changed to `failed`, the suspicion is only "made official" when it is persistent enough, i.e. after k_f test epochs.

Likewise, when the site is `recovering`, it is "quarantined" for k_r test epochs before the suspicion of recovery to the `alive` state is disseminated (lines 40-42).

Partition Healing

Partition healing is attempted whenever the symptoms provided by diagnosis of the Global Suspicion Matrix indicate partitioning, upon which the state variable *partitioned(GSM_u)* is set (line 45). Rerouting of traffic for the partitioned sites is attempted, by resorting to the help of "friendly" sites in the domain. Function *reRoute(GSM_u)*, whose details we discuss below, is invoked for that purpose (line 46).

Partitions are sometimes due to temporary flickering of the network, and thus short-lived, but nevertheless disturbing, given the inertia of current IP routing. However, most of the times, they result from failure of a link, in a situation where there are alternate paths. However, for several reasons, routing in most global network settings, such as the Internet, is rather static, and not bound to change on account of these occurrences. However, the situation tends to change, namely in the forthcoming new Internet protocols. The *partition healing* facility is modular and can be disabled if and when standard protocols handle partitioning better.

The *partition healing* facility is performed on a best-effort basis by the Multipoint Network (MN) protocols running over IP, through IP-tunneling, under the control of the SF.

Let us define S_v, the local site, P_{out}, the *out-partition* (that is, the set of sites which S_v cannot communicate directly with), and P_{med}, the mediating sites, which can communicate with both S_v and sites in P_{out}. For example, imagine that *di.fc.ul.pt*, in Portugal (S_v), is cut off from *cornell.edu*, in the U.S.A. (belonging P_{out}), but *newcastle.uk*, in the U.K., (belonging to P_{med}, the mediator set), communicates both with *di.fc.ul.pt* and *cornell.edu*. Obvious places both for these syndroms to be detected and for the remedy to be applied are the facility gateways.

The idea is very simple. The SF, upon deciding 'partial-partition' failure, does not inform the SM module, as would be normal. Instead, it makes a call (function *reRoute(GSM_u)*) to the MN module, with the following semantics:

- **Reroute(**$S_p \in P_{out}$**,** $S_m \in P_{med}$**):** Reroute all communication from S_v addressed to site S_p in P_{out}, through site S_m in P_{med}

In the skeleton protocol, we chose to maintain this rerouting during k_h epochs (line 46). If the partition disappears in the meantime, state variable *healed(GSM_u)* is set, which makes the SF instruct the MN to go back to the old route (function *reRoute(GSM_u)* in line 51). However, the implementation of *reRoute* only allows the change back after k_h epochs of rerouting have elapsed. This is a very simple form of hysteresis, in order to avoid routing instability. In our example, traffic from *di.fc.ul.pt* to *cornell.edu* would then be temporarily tunneled to *newcastle.uk*, which would then forward it to *cornell.edu*. Details of the implementation of rerouting are given in [19].

Decision Functions

When a suspicion is made public (line 17), a decision function on GSM_u is invoked (line 18). This function implements the semantics of failure/recovery detection, and triggers a normally distributed decision process amongst the relevant sites, which exchange their GSM's and try to reach agreement on a common Global Suspicion Vector, GSV. If there is not sufficient evidence for the decision function to converge, the decision fails. Typically, when there is a failure, the GSM progressively stabilizes in the measure that all sites become aware of it. The decision function itself may assume several forms, as we have already seen. It is not always required that all SF's decide, that depends on the semantics. When it converges, into a Global Suspicion Vector, GSV_u, a notification of failure or recovery is issued to other modules, by sending them the new Global Suspicion Vector (line 26). An LSV can be re-

quested explicitly, for example if a site is suspected. Whenever an LSV_v is not trusted, for example when it is not returned after requested, the corresponding row in the GSM is filled with 'don't cares', X.

5.5 Participant Failure Detection

The Participant Failure Detector (PF) works very simply by exporting a set of pre-defined test types, to be used by applications and protocols for participant surveillance. The actual mechanisms are not presented in this paper, but they act both at the level of the Activity Support (see Figure 6) on the flow to and from the participants, and by using operating system probes. Examples of measurable symptoms are: no observable response, process crash signals from O.S., excessive length of input message queue, etc. Participant failure detection is performed in a way similar to site failure detection, though made exclusively locally. Likewise, the detector accepts requests to test specifications, from simple crash failure to more complex QoS failure, based on the test types exported.

The main data structure of a PF, as shown in Figure 6, is the *Probe Scratchpad*, P_{sp}. The P_{sp} contains the participant tests criteria and the Local Detection Vector (LDV), a vector of Booleans, with $\#\{\mathcal{P}\}$ positions, one for each local participant, such that $LDV[j]$ contains the state of P_j, as seen by the local Participant Failure Detector, PF. Whenever a participant P_j fails a test, $LDV[j] = 1$, and the PF reports the failure detection to the Participant Membership module (PM). Some of the detection improvement mechanisms discussed in Section 5 can be applied to participant failure detection as well, although we do not address it for simplicity.

As said before, this local detection is a reliable one, in comparison to site failure detection, which may be unreliable due to the FLP impossibility result. On the other hand, the failure of a site implies the failure of all participants residing there. We will take advantage of this fact in our 2-tier membership structure in the next section. In consequence, we define *participant failure* detection:

A participant P_f^u residing in site S_u is detected failed at site S_v iff:

- $u = v$ and $LDV[f] = 1$ by the local PF
- $u \neq v$ and site S_v detects site S_u as failed

6 Membership

The membership structure in NAVTECH has a few attributes, devoted to handling the scale and uncertainty problems of large systems:

modularity - separated from failure detection;

data independence - separated from data flow, hooked at specific points (taps);

two-tier - hierarchical membership structure, divided in site and participant levels;

sticky membership - group elements do not lose membership when they fail

Modularity and data independence are key for handling autonomous operation of failure detection, the membership structure itself, and the data flow modules. The taps are points with a well identified interface to the communication and activity protocols, where the membership structure can exert different kinds of synchronization and control.

The *two-tier structure* has several advantages. Firstly, when more than one participant joins the same group at a site, the joins subsequent to the first are lightweight, since the hard part of membership maintenance resides with the site membership part. Likewise, the same happens when participants leave the group, until the last one. Mechanisms trying to achieve a similar functionality were devised in DELTA4[31] and in ISIS[20], under the name of lightweight groups. NAVTECH casts the hierarchy in the architecture, widening the horizon for scalability. For example, several AS modules may exist and map onto the site CS module. However, in each AS module, genuine lightweight groups may further be mapped on regular participant groups[35]. The two-tier membership is also compatible with the separation of concerns between communication (CS) and activity (AS) pioneered in the NAVTECH architecture. The same site membership semantics may serve different participant membership semantics in the same system, according to the application needs. For example, on the same view-synchronous site membership, a primary-partition database replica management group and a designated partition[10] collaboration awareness group may coexist.

The *sticky membership* idea emerged from the work in the BROADCAST project and was followed by several systems designed in the project[5], including NAVTECH . The intuition behind it was that in situations of instability, typical of large-scale settings, frequent group changes would occur. However, most of those changes might be due to partitioning, and quite a few of them (e.g. mobile environments) to temporary, short-lived crashes. In order for the system to operate efficiently, two objectives should be sought: (a) the operation during those glitches should adapt to the situation; (b) the cost of re-insertion should be reduced.

Separating between *membership*, a long-lived status of group elements achieved by explicit registration, and *view*, the current state of a group defined by the reflexive relation of "not detected failed", proved useful to pursue those objectives. Examples of applications relying on this duality can be found in [4, 10]. When members suddenly depart from groups, because of crash or partitioning, they stick to the membership. However, the view is updated to exclude them. This information is precious for dynamic adaptation algorithms, for example, dynamic majority replica management. On the other hand, it eases the task of merging and/or recovery of members, based on the analysis of the configuration of membership and view that occurs upon a re-join. The Site Membership module, SM, handles site failure detection/recovery on behalf of site-groups, inasmuch as Participant Membership, PM, handles participant failure detection/recovery on behalf of a participant group. In consequence, from now on it should be clear that when we say "site S_u detects S_w as failed" or "participant P_q believes P_r to be alive", we mean the respective membership modules (based on failure detection information).

6.1 Membership Definitions

Participants become senders or members of a group, $G_i = \{P_q, P_r, P_s, ...\}$, or abandon it respectively through *join* and *leave* operations. A group is defined by a tuple $G_i \equiv \langle N, M, V \rangle$. The *name* of the group, $N(G_i)$, is the unique logical designation or logical address of the group. The *membership* of the group, $M(G_i) = \{P_q | P_q \in G_i\}$, is formed by the participants that joined it as *members*. Note that participants that joined as *senders* (denoted by the symbol $P_q \ni G_i$) are thus treated with less overhead. The *view* of the group is formed by the members that are mutually believed to be alive, $V(G_i) = \{P_q | P_q \in G_i \wedge alive(P_q)\}$. That is, for any P_q, P_r in the view, P_q satisfies the relation $alive \equiv$ *not considered failed by* $P_r, \forall r \neq q$.

A site-group, $SG_i = \{S_u, S_w, S_w, ...\}$, is defined by a tuple $SG_i \equiv \langle N, M, V \rangle$. The *name* N of the site-group is the same as the name of the group. The *s-membership* of a site-group SG_i, is the set of sites hosting all participants having entered group G_i, either as *senders* or as *members*, $M(SG_i) = \{S_u | S_u \ hosts \ P_q, \forall (P_q \ni G_i \vee P_q \in G_i)\}$. The *s-view* of the site-group, is formed only by the sites hosting *members*. The reason being that these are the *recipient* sites, to be consistent with our unidirectional group model. Furthermore, the view only contains the sites that are mutually believed to be alive, $V(SG_i) = \{S_u | S_u \in SG_i \wedge S_u \ hosts \ P_q \in G_i \wedge alive(S_u)\}$. In the context of NAVTECH , 'alive' for a site means "with connectivity", that is, for any $S_u, S_w, w \neq u$, in the s-view, S_u satisfies the relation $alive \equiv$ *not considered failed by* S_w.

By the sticky membership attribute, group and site-group membership survive failures while there is at least one member alive, and they are only modified by explicit requests. Group and site-group views are modified by unsolicited events, such as failure and recovery detection.

6.2 Site Membership

Each time a participant joins a group, the relevant communication endpoint is opened at the site level. The set of communication endpoints forms the *site-group*. The Site Membership module, SM, is concerned with the management of site-groups. Sites enter a site-group by means of *open* or *attach* requests, depending on whether local users will receive or send messages. This interface materializes the unidirectional group model of NAVTECH : the return of an attach is a sending interface, the return of an open is a receiving interface; whenever local participants of a group need both to send and receive, an open and an attach are made to the same site-group.

The main data structures of SM, as shown in Figure 6, are the *Site-Group Membership*, and the *Liveness Criteria*. The Membership structure holds all site-group membership and views. The Liveness Criteria structure is defined on a per-group basis, and holds the behavior definitions stipulated for failure detection in each site-group, for example, QoS specifications (see Section 5.2). The SM, based on the former parameters, instructs the SF to run failure detection tests. A liveness criterion is defined when a site-group is created, but can be modified during its operation.

6.3 Participant Membership

The innovative issue that we stress again at this point, is the separation of site and participant membership in NAVTECH . Participant membership concerns the genuine participant groups, or just *groups* from now on. Its goal is to assist participants in working together, namely in forming groups and managing their activity. Site membership concerns *site-groups*, which are a mapping of groups on the site structure, and its aim is to assist inter-site communication on behalf of group participants. Both objectives have traditionally been aggregated to the *Group Membership Problem* in the generality of other works [13, 23, 33].

The main data structures of a PM, as shown in Figure 6, are the *Group Membership*, and the *Activity Criteria*. The Group Membership structure holds the membership and views of all groups. The Activity Criteria are defined on a per-group basis, when the group is created. They establish the rules to follow when there is a change in the state of the group (due to failures, joins or leaves), concerning the automatic control of the group's activity by the PM, relieving the application from that problem. NAVTECH allows the definition of criteria other than the canonical primary partition membership.

The PM modules handle participant failure detection, which can be provided by the following means: participant failure detector (PF) report to the PM on the failure of a local participant; remote PM report to peer PMs on the failure of a remote participant; site membership (SM) report to the PM on the failure of a remote site— the reverse mapping is made, and all the remote participants residing there are reported failed.

6.4 Membership in Action

In order to clarify our membership structure, we are going to highlight a few fundamental aspects of its functionality. We start by doing an overview of the mapping between the PM and SM solicited actions of entering and leaving groups and site-groups, and explain what is gained from a 2-tier, sticky membership structure:

- **First participant join -** the first participant to join a group at a site must create the communication endpoint for the corresponding site-group: it invokes an open (member), an attach (sender), or an open+attach (sender/member), and in response it receives a receive or a send endpoint, or both.
- **Site-group open/attach -** the site enters a communication group, performing the necessary actions to stabilize and achieve consensus on the site-group membership and view including the new site, accordingly to the chosen semantics (e.g. view synchrony); attach is supposedly simpler than open.
- **Sequel of first participant join and following joins -** in the 2-tier membership, synchronization of the participant's entry and update of the membership and view to include the newcomer, accordingly to the chosen semantics for this group, can be performed by merely having the PM use the communication channel to the group (e.g. an atomic multicast message will ensure total order w.r.t. to messages exchanged in the group); a further advantage is that the synchronization of further joins is as simple as increasing a counter of group users at the site, and synchronizing the entry as above.
- **Participant leave -** the user counter is decreased, and the PM synchronizes the departure with the other group members, easing its removal from the membership and view; upon the last local leave from that group, the site also leaves the corresponding site-group.

- **Site-group close/detach -** the site leaves the site-group by signalling its departure to the other member sites, so as to ease stabilization of the new site-group membership and view without this site; the endpoint is closed.

Failure, Partitioning and Recovery

The power of the 2-tier structure with sticky membership is also evident upon the occurrence of unsolicited actions, such as failures, recoveries, partitioning, and merging. We enumerate below the actions triggered upon failure or recovery of sites and participants. Since a participant or a site may either fail or be partitioned (vis. recover or merge) without the distinction being evident at first sight, we use up and $down$ to denote either state, from a detector's perspective.

- **Participant down -** when a participant in the membership is down, it is removed from the group view — $down(P_q \in G_i) \Rightarrow V(G_i) = V(G_i) - P_q$
- **Site down -** when a site in the membership is down
  it is removed from the site-group view — $down(S_u \in SG_i) \Rightarrow V(SG_i) = V(SG_i) - S_u$
- **Site up -** when a site in the membership is up again it is added to the view —
  $up(S_u \in SG_i) \Rightarrow V(SG_i) = V(SG_i) + S_u$
- **Participant up -** when a participant in the membership is up again, it is added to the view — $up(P_q \in G_i) \Rightarrow V(G_i) = V(G_i) + P_q$

Because of the sticky membership and membership/view duality, when a site S_u shows up to an existing site-group SG_i, the information provided both by its structures for that site-group ($M^u(SG_i)$ and $V^u(SG_i)$) and by the other sites' structures ($M^v(SG_i)$ and $V^v(SG_i)$) allow the identification of the site's state and intention. Observe that in a system of uncertain timeliness and prone to partitioning, sites may change their state with regard to a site-group, even in the middle of a protocol execution, in the course of partitioning. For example, consider site A has sent a message to site B and now waits for B's acknowledgment. It may happen that the next time A hears about B, B and others that in the meantime had formed a virtual partition excluding A, due to connectivity problems, are now coming back because connectivity improved in the meantime: sites A and B have contradictory membership/view information. However, as we show below in the table of Figure 8, NAVTECH provides useful information for protocols to detect the actual situation of a site that shows up to an existing site-group: cold start; reboot, with or without memory; merger; connected. This helps automate and modularize the site insertion protocols and procedures, in order to provide quick reaction to attempts by sites to enter existing site-groups. As a matter of fact, this is as far as we wish to go in the NAVTECH framework: leaving generic and well-defined hooks for specific cold start, recovery and merging algorithms to be implemented.

Activity Criteria

With the notion of Activity Criteria, the NAVTECH framework provides a hook for the definition of generic criteria to control activity of subgroups in the presence of partitioning. These criteria may be different from the canonical example of *primary partition, PP*, and they can vary from group to group.

Type of Recovery	Description
boot	membership information about the site is non-existant $S_u \notin (M^u(SG_i), M^v(SG_i)), \forall v$
stateless reboot	the site does not have membership information itself, but it appears in the membership information of the other sites $S_u \notin M^u(SG_i) \wedge S_u \in M^v(SG_i) \wedge S_u \notin V^v(SG_i), \forall v$
stateful reboot	the site has kept membership information in non-volatile storage, and appears in the membership of the other sites $S_u \in M^u(SG_i) \wedge S_u \notin V^u(SG_i) \wedge S_u \in M^v(SG_i) \wedge S_u \notin V^v(SG_i), \forall v$
merger	the site has its own membership information, and consistently appears in the membership but not in the view of the other sites $S_u \in (M^u(SG_i), V^u(SG_i)) \wedge S_u \in M^v(SG_i) \wedge S_u \notin V^v(SG_i), \forall v$
connected	when a site is fully connected, membership information about the site is consistent everywhere $S_u \in (M^u(SG_i), V^u(SG_i) \wedge M^v(SG_i), V^v(SG_i)), \forall v$

Figure 8: Identification of the state of a recovering site

The activity criterion of a group is defined as a boolean function of the membership and view of the group, $AC_i = f(M(G_i), V(G_i))$. An *Activity test*, performed whenever a local PM module handles a change in the group state, evaluates the function for a given pair membership/view, and if it evaluates to false, activity is blocked for all participants at that site.

An example function for the primary partition criterion is $AC_i \equiv V(G_i) > M(G_i)/2$, that is, activity blocks for participants of group G_i that find themselves in a minority partition, since AC_i evaluates to false at the relevant PM modules. This happens in this case because upon partitioning, some sites formed a subgroup such that the view $V(G_i)$ does not reach majority. However, there are a number of other, significant activity control criteria in real world systems. For example, a minimum absolute threshold on the number of participants, $AC_i \equiv V(G_i) \geq N$, typical example of a manager of a distributed processor pool, where N workers are assembled to perform a distributed parallel computation, and where despite partitions or failures, what is relevant is that there are N for each requested computation. Several partitions may work in parallel in this case, provided there are at least N participants. In cooperative applications, it is frequent to define an enumerated set of core named participants without which the application cannot proceed (e.g. the moderator, the chair, etc.)[10]. Suppose a virtual group decision meeting where in order to make progress in certain parts of the meeting, the moderator, plus at least three persons, must be present: such a criterion would look like $AC_i \equiv P_{mod} \in V(G_i) \wedge \#\{V(G_i)\} > 3$.

6.5 Two-tier Membership Skeleton Protocol

We now present the specification of a skeleton protocol that implements the membership management functionality. In the framework spirit of this work, we identified several generic functional modules materializing the steps we have just discussed, and organized the skeleton around them. This way it is possible to implement new membership protocols without having to re-invent modules that are common to most membership algorithms. In the work of [23] several functional modules are provided so that they can be configured in a number of possible ways. In comparison, our skeleton protocol adopts one such interaction and configuration semantics, the 2-tier sticky membership structure, because of the advantages in efficiency that we have just shown, namely for large-scale, uncertain synchrony settings. However, it provides placeholders where modules such as those developed in [23] can be re-used, so that different membership semantics can be implemented in NAVTECH .

The consolidated view of the transitions in membership is as given in the table of Figure 9. The symbol ($\rightarrow$) denotes "added to", whereas ($\leftarrow$) denotes "withdrawn from". As for the algorithm, it is presented in Figures 10 and 10, and in general, it makes use of the following notation: whenever the state of a group or site-group changes, a state variable $stateChangeXX(YY)$ is positioned; XX accounts for the module where the state change is produced: PM, SM, PFD or SFD; YY accounts for the structure concerned with the change: group or site-group.

Event	**Action on Membership/View**
participant join as member	$P_q \rightarrow M(G_i), V(G_i)$
participant join as sender	——
participant leave	$P_q \leftarrow M(G_i), V(G_i)$
site open	$S_u \rightarrow M(SG_i), V(SG_i)$
site attach	$S_u \rightarrow M(SG_i)$
site close	$S_u \leftarrow M(SG_i), V(SG_i)$
participant down	$P_q \leftarrow V(G_i)$
site down	$S_u \leftarrow V(SG_i)$
site up	$S_u \rightarrow V(SG_i)$
participant up	$P_q \rightarrow V(G_i)$

Figure 9: Membership and view change rules

Solicited Actions

The obvious solicited actions implemented by the skeleton membership protocol are join, leave, open, attach, close, detach. They prefigure the membership functions for entering and exiting groups and site-groups discussed in Section 6.4. If a request to join (lines 01 or 10) is the first at the site, it is necessary to enter the site-group first (lines 02 or 11), which returns a receive or transmit handle for the group, depending on whether an open or an attach are made, respectively. Depending on whether a participant joins as member or as sender, a receive or a transmit handle are returned (lines 04 or 12). We leave the analysis of leave, close and detach operations, to the reader.

Synchronizing Group Views

The synchronizer protocol (*synchronize*($P_q, action, G_i$)) synchronizes the members of group G_i to carry a change of state due to an action (join,leave,down,up) performed on P_q in G_i. This may be done in several ways that we do not discuss here, according to the chosen semantics (ordering, state transfer, etc.). For example, a totally ordered entry may be performed simply by issuing an atomic multicast message to the group. When the function returns, all members will have updated $M(G_i)$ and $V(G_i)$, including P_q, depending on the action, as per Table 9.

```
      // ** PM **
      // ** solicited actions
01 when join(G_i,member) do
02    if ∄SG_i then RXhandle=open(SG_i) fi
03    synchronize(P_q, join, G_i)
04    return RXhandle
05 od
10 when join(G_i,sender) do
11    if ∄SG_i then TXhandle=attach(SG_i) fi
12    return TXhandle
13 od
15 when leave(G_i,member) do
16    synchronize(P_q, leave, G_i)
17    if lastmember(SG_i) then close(SG_i) fi
18 od
20 when leave(G_i,sender) do
21    if lastsender(SG_i) then detach(SG_i) fi
22 od

      // ** change events
25 when stateChangePM(G_i) do
26    if testActivity(G_i)
27       then action=unblock(G_i)
28       else action=block(G_i) fi
29    updateAT(V(G_i), action)
30 od
35 when stateChangePM(SG_i) do
36    update(G_i)
37 od

      // ** unsolicited actions
40 when stateChangePFD(P_q,up|down) do
41    if P_q ∈ G_i then
42       synchronize(P_q, up|down, G_i) fi
43 od
```

```
      // ** SM **
      // ** solicited actions
50 when open(SG_i) do
51    stabilize(S_u, open, SG_i)
52    return RXhandle
53 od
55 when attach(SG_i) do
56    stabilize(S_u, attach, SG_i)
57    return TXhandle
58 od
60 when close(SG_i) do
61    stabilize(S_u, close, SG_i)
62 od
65 when detach(SG_i) do
66    stabilize(S_u, detach, SG_i)
67 od

      // ** change events
70 when stateChangeSM(SG_i) do
71    updateCT(V(SG_i))
72 od
      // ** unsolicited actions
75 when stateChangeSFD(S_u,down) do
76    stabilize(S_u, down)
77 od
80 when stateChangeSFD(S_u,up) do
81    stabilize(S_u, up)
82 od
```

Figure 10: Two-tier Membership Skeleton Protocol – PM and SM (For each S_u at any P_q,S_u)

Stabilizing Site-group Views

The stabilizer protocol (*stabilize*($S_u, action, SG_i$)) processes solicited actions that change the state of a site-group. It stabilizes the outcome of solicited actions (open,attach,close,detach) on S_u in site-group SG_i. This is done according to the chosen semantics for managing site-

group membership (e.g. flushing and view-synchrony or partial order, strong or weak partial, etc.). For example, with view-synchronous semantics, the stabilizer takes care of flushing all pending messages in the site-group, reach agreement among the site-groups on the next membership and/or view, and install it everywhere. When the function returns, all sites will have updated $M(SG_i)$ and/or $V(SG_i)$, including S_u, depending on the action, as per the table in Figure 9. Attach is expected, in most implementations, to lead to simpler actions, and it only implies the update of $M(SG_i)$.

The stabilizer also stabilizes the outcome of unsolicited actions (down,up) of S_u in all the site-groups that contain S_u (*stabilize(*$S_u, action$*)*). The function handles failures/recoveries efficiently by returning results for all site-groups affected, as if invoked for one site-group. In the particular case of recovery (up), the stabilizer protocol extracts information as described in the table of Figure 8, to identify the type of recovery of the site for each site-group it belongs to, and perform the adequate actions, as discussed in Section 6.4.

Testing Activity Criteria

Whenever the state of a group changes, for example in result of the synchronization functions just described, a state variable $stateChangePM(G_i)$ is positioned (line 25). This triggers the evaluation of the activity criterion for that group, that blocks or unblocks activity of this group at this site, depending on whether it evaluates to false or not (lines 26-28). We made these operations idempotent for simplicity. If blocked, all subsequent local requests to this group are refused.

Updating the Taps

The information about site view changes must be propagated to the communication protocols. This is done through the Communications Tap (*updateCT(*$V(SG_i)$*)*), which is the hook to the data flow part to update the view of SG_i (lines 70-71). Likewise, group view changes must be propagated to the activity protocols. This is done through the Activity Tap (*updateAT(*$V(G_i), action$*)*), which besides to the update of the view, also does the blocking or unblocking of G_i ($action$), as a result of the activity test (line 30).

Detecting Failures

A participant failure detected locally must be disseminated to the rest of the group, with the same synchronization semantics used for solicited operations (lines 40-42). The implementation of the synchronizer protocol should handle that. Subsequently to the change in view, activity is tested and the AT updated (lines 25-30).

A site failure entails the run of the stabilizer protocol over the whole domain, with the aim of reaching fast agreement on the new view, at all sites (lines 75-76), *for all site-groups* affected by the failure. When each of the concerned site-groups is detected as changed (line 70), the CT is updated.

Site failures and recoveries must be propagated to the PM. Since this is normally required to be ordered with the data flow, it is done by propagating the information from

the CT (line 71) to the AT through the data path. The arrival of the information to the AT causes a state change at the PM (line 35) which is processed by making updates in the group connected to the changed site-group (line 36).

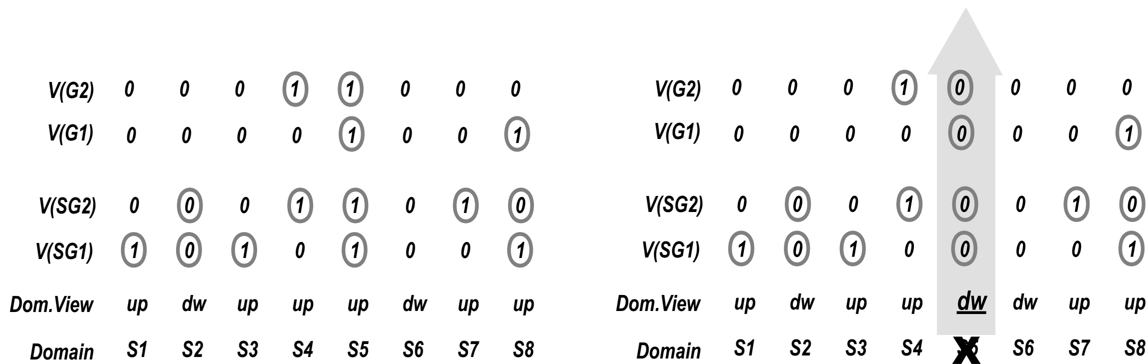

Figure 11: Propagation of failure information through the 2-tier membership structure

Recovering

When a recovering site is detected, the stabilizer protocol is run for recovery (lines 80-81), in the domain, for the same reasons as for failure detection. However, recovery of sites offers a number of opportunities for optimized protocols to be built. NAVTECH goes as far as providing those protocols with information about the kind of recovery, see the table of Figure 8. The stabilizer may be built so as to react differently, depending on whether it is a: cold start; reboot, with or without memory; merger. In a cold start, nothing is done, the site is allowed to go up, launch the applications, that will eventually open their groups. In a reboot without memory, the site-groups may be activated automatically, and the application is allowed to a warm restart with the communication channels already open. In a reboot with memory, it is possible that the application may enjoy some stable storage backup and do a warm restart with the support of the system. In a merger, the merging sites enter a negotiation to recognize what is the correct merged state. After stabilization, the relevant site-group views are changed: CT is updated (line 71), and that information is propagated up to the AT. In consequence, when the AT reports a site-group SG_i state change to the PM (line 35), the relevant group G_i is updated in consequence. Figure 11 exemplifies with a bit mask example how efficient the management of failures and recoveries may be, if algorithms take advantage from the propagation path through the 2-tier membership hierarchy.

7 The Data Flow Modules

Remember that the membership and failure detection structure are the spinal cord of NAVTECH . As depicted in Figure 6, the *taps* act on the data flow, performing filtering, multiplexing and demultiplexing functions, allowing to implement several known functionality and consistency profiles.

At the Multipoint Network level, the bare semantics of the site failure detector SF enforces an informal weak-partial membership in the best-effort multicast datagram commu-

nication protocols. The hook is the Multicasting Tap. In this sense, several possibly overlapping views may be formed in a system, namely in the presence of partitioning. Those views reflect in the best manner possible the set of sites than are mutually reachable, but no further guarantees are given. This is the support of the *best-effort* profile, where the informal views are mainly supposed to assist multicast addressing and best-effort communication, as implemented by the Multipoint Network. The best-effort profile can be connected to protocols further up, but it bypasses all other tap filters up to the participant, as sketched in Figure 5. This profile has been tested in a prototype of a partitionable cooperative application[10].

Note that NAVTECH combines two desirable but sometimes conflicting attributes: (a) semantic guarantees (e.g. view-synchrony, primary-partition); (b) versatility (tap bypass, group composability). From our past experience with group systems, we believe this was made difficult for two reasons: the functionality of membership and failure detection was mixed with communication; the structure of membership and failure detection was intertwined with the data flow protocols.

The first syndrome has been addressed in practically all recent groups systems, by separating the membership and failure detection functions from communication. However, most of the current groups systems, although modular, still have membership stand "in the way" of the data flow. Suppose that site and participant membership would be inside of or closely coupled to the CS and AS modules respectively: it might be difficult to reason about the correctness of applications that would for example rely on some communication protocols in CS, bypass the primary-partition semantics, and use again some protocols of the AS module, for example, session control and awareness in a CSCW application.

Now consider the same application in NAVTECH , and suppose it was a cooperative computer supported application. The architect might define liveness criteria for the site connectivity based on QoS information, and hook them to the Communications Tap. Then he might define activity criteria based on alternative partition semantics, such as the social roles of participants, and hook that to the Activity Tap. Then he would connect the necessary protocols at the different levels to configure the application communication and processing needs, without worrying about consistency, ensured by the membership/failure detection structure, through the tap hooks, which change at precise steps in the execution.

We now review the several data flow modules: Multipoint Network, Communications Services, Activity Services, with emphasis on the Tap functionality.

The Multipoint Network

Sites are administratively *registered* in the Domain. Each local Site Failure Detector has a view of the currently reachable sites in the Domain, the *Domain View*, C_{sp}, kept in the *Connectivity Scratchpad*, as was shown in Figure 6. As previously said, this view may be incomplete, but must at least contain the set of all sites containing site-group members.

This view is updated by the failure detection protocols, and assists the addressing of the low-level multicasting protocols, through the Multicasting Tap. The Domain may obviously be very large, depending on the scale of the applications. In consequence, the scope of inter-site communication for failure detection may be subdivided using the Facility-based hierarchy. This topology awareness suits the protocol functionality, but is kept transparent

to its semantics.

The Multipoint Network presents two classes of interfaces: *LocalNet* and *GlobalNet*, as shown in Figure 6. The *GlobalNet* is the inter-Facility communication. A Facility resides in a LocalNet. The *LocalNet* is the intra-Facility communication. NAVTECH is built on top of current standard network architectures and protocols, for example, but not limited to: bridged Ethernet for the LocalNet; the Internet for the GlobalNet.

The LocalNet and GlobalNet protocols are implemented on top of the relevant standard protocol drivers, and traffic is routed to/from either by a dispatcher. The dotted lines in the figure illustrate the data paths. A possible implementation of GlobalNet addressing is through internet group addressing such as multicast-IP, addressing all the Agents of Facilities holding addressed sites. A possible implementation of LocalNet addressing is through physical group addressing, such as Ethernet multicast.

Communication Services

The Comunication Services module hosts the communication protocols and any other site-related protocols (such as clock synchronization). The CS is designed in order to be implementable by any runtime environment that is modular enough to be stripped from site membership, and in consequence be hooked downwards to the Communications Tap, as a provider of the desired membership semantics. The runtime environment is hooked upwards to the dispatcher, which hides the fact that the CS may be serving several Activity Services modules, probably in different address spaces.

Figure 6 suggests the use of protocol composition in the implementation of NAVTECH protocols. Although the architecture itself is not tied to a concrete protocol composition framework, its implementation should desirably rely on a run-time environment that allows micro-protocols to be composed in a flexible way. Typical examples of such frameworks are the x-kernel[29], Horus [40], Ensemble [22], Coyote [6] and *Appia*[25]. The lessons learned from these implementations motivated the development of a new composition framework[30]. This framework, called *Appia*[25], is currently being used to implement an instantiation of the NAVTECH architecture to support intrusion-tolerance techniques[9]. The advantage of *Appia* over previous composition frameworks is that it allows different communication channels, each with its own set of services, to be integrated in a coherent multi-channel protocol stack. *Appia* recognizes the need to integrate channels, allowing properties to be shared across several channels, a feature that is extremely useful in the implementation of modular architectures.

Communication Tap

The SM is hooked with the CS module by the Communication Tap, CT. The CT acts as a filter to enforce the SM strategies, by controlling the communication flow:

- it diverts incoming protocol messages destined to the SM, albeit preserving their order relative to the data message flow;
- securing s-membership properties in the communication flow between site-group members;
- informing upper layers about site-group view changes;

- blocking communication when necessary to enforce the group communication properties (e.g. view-synchronous flush);
- by multiplexing/demultiplexing, the CT also maps the communication flows of the several site-groups onto the MN.

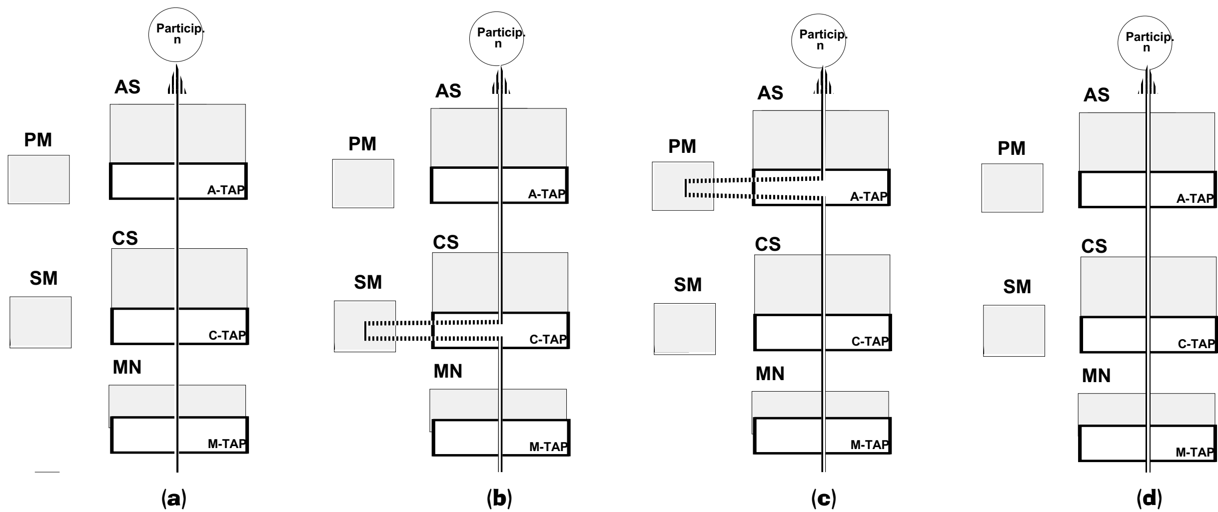

Figure 12: Taps in action

The operation of the taps is exemplified in Figure 12. On the left (a), the normal operation, taps let the flow pass between the MN and PM. Next to the right (b), a change is detected in the site level. CT tap diverts the flow to the SM module, for the change to be processed and the new view stabilized by the SM protocols (see Section 6.4), in complete control of the desired semantics w.r.t. the data flow. The following picture (c) shows the result of the propagation of the update to the AT tap: the tap diverts the flow through the PM module, so that the resulting changes may be processed, by having them reflected in the participant group view, and this view synchronized among all participants, with a controlled semantics w.r.t. the data flow.

Activity Services

The Activity Services (AS) host the algorithmic part of the distributed application support. In terms of runtime, the considerations we made for the Communication Services apply as well for AS. That is, the same kind of runtime environment can be re-instantiated for AS, hosting algorithms as modular as the communication protocols, as portrayed in Figure 6. However, the function of AS is supporting the execution of distributed applications, leaving communication to CS. In consequence, this is the place for replication management, coordination and cooperation algorithms. This modularity, and the fact that several AS modules may be instantiated over the CS module, leaves room for the deployment of different middleware support susbsystems re-using the same communication services: fault-tolerant toolboxes, CSCW platforms, etc.

Activity Tap

The PM connects to the AS module by the Activity Tap, AT. The Activity tap establishes a hook between participant membership (PM) and activity management protocols (e.g. replication management, collaboration awareness). It also implements the desired semantics for group membership, based on the underlying s-membership semantics. For example, view-synchronous s-membership allows securing a linear membership at the participant level, that is, one with a totally ordered history of views. With such a semantics, applications on top may enjoy the properties characterizing the *primary-partition* profile, namely, based on the majority criterion.

The AT, as a filter, is the means of enforcing PM-derived semantics, by inserting control information in the correct places of the communication flow from/to the participants:

- securing membership properties in the flow of activity among participants;
- informing the participants about view changes;
- blocking or unblocking the activity flow according to the group activity criteria;
- by multiplexing/demultiplexing, the AT also maps several participants onto a same group.

Program-controlled Failure Notification

In all previously known systems, the failure detector, besides being a crash-only detector, is directly hooked to the group membership protocol. We introduced *program-controlled failure notification* in NAVTECH , whereby the notification of the failure of S_i may be prevented or produced by the application upon the analysis of failure notifications (e.g. results of QoS tests). This is possible when programming on top the best-effort profile, where the traditional chain SF-SM is broken. The site failure detector reports directly up, as we can see in Figure 5. The Tap filters are bipassed, and the membership services in this case, work on a best effort basis, to maintain a notion of group whose semantics is directly controlled by the application, or by adequate Activity Services, such as the NAVCOOP platform for support of cooperative applications[10]. These services receive failure/recovery notifications, analyze them, and decide whether and when to pass the notification to membership.

Note that letting high-level services (or even the application) gain control over failure detection is a risky decision, probably only feasible because our failure detector does less mistakes and provides richer information (QoS). Mistakes on account of environment instability are significantly lower with an analog decision criterion (QoS failure detection), and some dampening (suspicion debouncing, partition healing).

QoS Adaptation

This crucial architectural feature was used by applications built on top of NAVTECH 's best-effort profile, to improve the accuracy of failure detection and avoid premature group exclusions by performing QoS adaptation[10]: we did an on-line comparison of the decisions of our QoS failure detector (QoS-FD) with those of a classical crash failure detector, in a test-bed where the environment was degraded sporadically by fault injection, and indeed the QoS-FD showed significantly more stable operation.

The way it works is as follows. Recall the process of QoS failure detection (Section 5.2):

at the end of each epoch, each site has information about the connectivity to other sites. Normally, in case of a failure, that is, when $LSV_u[v] = 1$ after a test epoch, the site failure detector (SF) should go on and pass the decision to the Site Membership (SM) module, which would automatically trigger a group change. However, remember that at the end of each epoch, each process S_u has thorough information provided by the QoS-FD about the QoS of its connectivity with other sites, namely the values of all parameters and DI_u^v.

This information allows two possible outcomes: either the intended QoS-dependent computation is not possible at this time, or at least involving the faulty site(s), and failure is then declared to the SM; or the application can fine-tune parameters, for example by choosing which ones to relax when DI shows insufficient QoS, and remain active, avoid premature failure declaration. The level of demand of the QoS specification for the next epochs is relaxed, and the group resorts to a degraded mode of operation, maintaining all members. Likewise, when a parameter P holds with 100% coverage, the application may tighten the specification. This game of tightening some and loosening others aims at the final end-to-end goal, that of obtaining the best possible QoS.

8 Conclusions

Making few or no assumptions about the network infrastructure ensures high portability but often yields disappointing performances. The successful design should capture the right balance between generality and performance. In the NavTech project, we experimented with a global network model that we have called WAN-of-LANs. The model is simple and general, and can be applied to most existing global infrastructures. Yet, it is powerful enough to allow the protocol designer to exploit its hierarchical nature to improve the performance or reliable communication protocols.

There are a few published protocols exploiting the innovative attributes of NavTech , namely topology and group model. For example, a lightweight group service[36] and a reliable remote group access protocol[37], both taking advantage from the sender/member and site/participant duality of the group model, and benefit from the topology to have credible efficiency. Another protocol is a clock synchronization protocol based on an hierarchical composition of a protocol tailored to broadcast LANs with a protocol that makes use of the GPS architecture[45]. The fourth protocol is a causal order protocol that makes use of topology information to reduce the size of information that needs to be stored and exchanged to provide causal order delivery[38]. Finally, we refer the reader to a total order protocol that dynamically adjusts the ordering algorithm as a function of the participant location in the network topology[34]. All the solutions follow the general framework provided by the NavTech architecture. The experience with these protocols reinforced our belief that the model is appropriate to design efficient group-oriented systems for large-scale networks.

Acknowledgments

Luís Rodrigues designed the generic remote invocation protocol, the light-weight group service and several topological communication protocols, such as the topological causal order

algorithm and the hybrid total order algorithm. He and António Casimiro were also involved in the design of the CesiumSpray clock synchronization service. Henrique Fonseca took part in the work on total order protocols, and also provided simulation results for the causal order protocols. A prototype implementation of the abstract network was done by Jorge Frazão. He also performed part of the work on QoS failure detection, specified by François Cosquer and Luís Rodrigues. The former used it to provide QoS-adaptability of cooperative applications in the NavCoop platform. A prototype of a group membership protocol was implemented by António Sargento. Werner Vogels participated in the early definition of the network model. We owe our colleagues in the BROADCAST Esprit Project many challenging discussions on the concepts presented here.

References

[1] M. Aguilera, W. Chen, and W. Toueg. Failure detection and consensus in the crash-recovery model. In *Proc. 12th Int. Symposium on DIStributed Computing (formerly WDAG)*, pages 231–245, Andros, Greece, sep 1998. Sringer-Verlar LNCS 1499.

[2] Carlos Almeida and Paulo Veríssimo. Timing failure detection and real-time group communication in *quasi-synchronous* systems. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, L′ Aquila, Italy, June 1996. (also available as INESC technical report RT/20-95).

[3] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high-availability. In *Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing Systems*, pages 76–84. IEEE, 1993.

[4] Özalp Babaoğlu, Alberto Bartoli, and Dini Gianluca. On programming with view synchrony. Technical Report UBLCS-95-15, University of Bologna, September 1995.

[5] Özalp Babaoğlu, Andre Schiper, and Paulo Veríssimo. Group communication in large-scale distributed systems, mar 1994. Extended Abstract (Cabernet Workshop, Dublin).

[6] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems*, 16(4):321–366, November 1998.

[7] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM, Transactions on Computer Systems*, 9(3), August 1991.

[8] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[9] M. Correia, P. Veríssimo, and N. F. Neves. The architecture of a secure group communication system based on intrusion tolerance. In *The 21st International Conference on Distributed Computing Systems Workshops. International Workshop on Applied Reliable Group Communication*, pages 17–22, April 2001.

[10] François J.N. Cosquer, Pedro Antunes, and Paulo Veríssimo. Enhancing dependability of cooperative applications in partitionable environments. In *Dependable Computing - EDCC-2*, volume 1150 of *Lecture Notes in Computer Science*, chapter 6, pages 335–352. Springer-Verlag, October 1996.

[11] François J.N. Cosquer, Luís Rodrigues, and Paulo Veríssimo. Using Tailored Failure Suspectors to Support Distributed Cooperative Applications. In *Proceedings of the 7th International Conference on Parallel and Distributed Computing and Systems*, pages 352–356. IASTED, October 1995.

[12] F. Cristian and C. Fetzer. The timed-asynchronous system model. Technical Report CS97-517, UCSD, January 1997.

[13] Flaviu Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.

[14] Flaviu Cristian, Robert D. Dancey, and Jon Dehn. High Availability in the Advanced Automation System. In *Digest of Papers, The 20th International Symposium on Fault-Tolerant Computing*, Newcastle-UK, June 1990. IEEE.

[15] P. Eugster and R. Guerraoui. Probabilistic multicast. In *Proceedings of the International Conference on Dependable Systems and Networks*, Washington, USA, June 2002, to appear. IEEE Computer Society.

[16] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 296–306. IEEE, 1995.

[17] Cristof Fetzer and Flaviu Cristian. Fail-awareness, an approach to construct fail-safe applications. In *Digest of Papers, The 27th International Symposium on Fault-Tolerant Computing*, Seattle - USA, July 1997. IEEE.

[18] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery*, 32(2):374–382, April 1985.

[19] Jorge Frazão. Enhancing large-scale communication with failure suspectors and dynamic routing (in portuguese). Master's thesis, Instituto Superior Técnico, Lisboa, Portugal, April 1995.

[20] B. Glade, K. Birman, R. Cooper, and R. Renesse. Light-weight process groups in the isis system. *Distributed System Engineering*, (1):29–36, 1993.

[21] I. Gupta, R. van Renesse, and K. Birman. Scalable fault-tolerant aggregation in large process groups. In *Proceedings of the 2001 IEEE International Conference on Dependable Systems and Networks*, Gotteborg, Sweden, June 2001. IEEE Computer Society.

[22] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, Computer Science Department, 1998.

[23] M. Hiltunen and R. Schlichting. Properties of membership services. Technical report, University of Arizona, Depratment of Computer Science, Tucson, AZ, August 1994.

[24] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental study of internet stability and backbone failures. In *Digest of Papers, The 29th Annual International Symposium on Fault-Tolerant Computing*, Madison, USA, June 1999. IEEE Computer Society.

[25] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 707–710, Phoenix, Arizona, April 2001. IEEE.

[26] Shivakant Mishra, Larry L. Peterson, and D. Schlichting. Consul: A Communication Substracte for Fault-Tolerant Distributed Programs. *Distributed Systems Engineering*, 1(2):87–103, December 1993.

[27] L. Moser, P. Melliar-Smith, A. Agarwal, R. Budhia, C. Lingley-Ppadopoulos, and T. Archambault. The totem system. In *Digest of Papers of the 25th International Symposium on Fault-Tolerant Computing Systems*, pages 61–66. IEEE, June 1995.

[28] J. Pereira, L. Rodrigues, and R. Oliveira. Reducing the cost of group communication with semantic view synchrony. In *Proceedings of the International Conference on Dependable Systems and Networks*, Washington, USA, June 2002, to appear. IEEE Computer Society.

[29] L. Peterson, N. Buchholz, and R. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–146, August 1989.

[30] A. Pinto, H. Miranda, and L. Rodrigues. Light-weight groups: an implementation in ensemble. In *Fourth European Research Seminar on Advances in Distributed Systems (ERSADS'01)*, Bertinoro (Forli),Italy, May 2001.

[31] D. Powell, editor. *Delta-4 - A Generic Architecture for Dependable Distributed Computing*. ESPRIT Research Reports. Springer Verlag, November 1991.

[32] Michel Raynal, Andre Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information processing letters*, 39(6):343–350, September 1991.

[33] Aleta M. Ricciardi and Kenneth P. Birman. Using process groups to implement failure detection in asynchronous environemnts. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 341–351, August 1991.

[34] L. Rodrigues, H. Fonseca, and P. Veríssimo. Totally ordered multicast in large-scale systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 503–510, Hong Kong, May 1996. IEEE.

[35] L. Rodrigues, K. Guo, A. Sargento, R. van Renesse, B. Glade, P. Veríssimo, and K. Birman. A transparent light-weight group service. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, pages 130–139, Niagara-on-the-Lake, Canada, October 1996.

[36] L. Rodrigues, K. Guo, P. Veríssimo, and K. Birman. A dynamic light-weight group service. *Journal of Parallel and Distributed Computing*, 60(12):1449–1479, December 2000.

[37] L. Rodrigues, Ellen Siegel, and P. Veríssimo. A Replication-Transparent Remote Invocation Protocol. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, Dana Point, California, October 1994. IEEE.

[38] L. Rodrigues and P. Veríssimo. Causal separators for large-scale multicast communication. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems (ICDCS'15)*, pages 83–91, Vancouver, British Columbia, Canada, May 1995. IEEE.

[39] L. Rodrigues and P. Veríssimo. Topology-aware algorithms for large-scale communication. In S. Krakowiak and S. Shrivastava, editors, *Advances in Distributed Systems*, LNCS 1752, chapter 6, pages 127–156. Springer-Verlag, 2000.

[40] R. van Renesse, Ken Birman, and S. Maffeis. Horus: A flexible group communications system. *Communications of the ACM*, 39(4):76–83, April 1996.

[41] P. Veríssimo, A. Casimiro, and C. Fetzer. The timely computing base: Timely actions in the presence of uncertain timeliness. In *Proceedings of DSN 2000, the IEEE/IFIP Int'l Conf. on Dependable Systems and Networks*, pages 533–542, New York City, USA, June 2000. IEEE Computer Society Press.

[42] P. Veríssimo and José A. Marques. Reliable broadcast for fault-tolerance on local computer networks. In *Proceedings of the 9th Symposium on Reliable Distributed Systems*, Huntsville, Alabama-USA, October 1990. IEEE.

[43] P. Veríssimo and M. Raynal. Time in distributed system models and algorithms. In S. Krakowiak and S. Shrivastava, editors, *Advances in Distributed Systems*, LNCS 1752, chapter 1, pages 1–132. Springer Verlag, 2000.

[44] P. Veríssimo, L. Rodrigues, and A. Casimiro. Cesiumspray: a precise and accurate global clock service for large-scale systems. *Journal of Real-Time Systems*, 12(3):243–294, 1997.

[45] P. Veríssimo, L. Rodrigues, and A. Casimiro. Cesiumspray: a precise and accurate global time service for large-scale systems. *Journal of Real-Time Systems*, 12(3):243–294, 1997.

[46] Paulo Veríssimo and Carlos Almeida. Quasi-synchronism: a step away from the traditional fault-tolerant real-time system models. *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, 7(4):35–39, Winter 1995.

[47] Paulo Veríssimo and Luís Rodrigues. Group Orientation: a Paradigm for Modern Distributed Systems. In *Proceedings of the 5th ACM SIGOPS European Workshop*, Mont Saint-Michel, France, September 1992. Extended and revised version as INESC RT/20-94.