Topology-aware algorithms for large-scale communication*

Luís Rodrigues and Paulo Veríssimo

Faculdade de Ciências, Universidade de Lisboa, Lisboa, PORTUGAL,
{ler,pjv}@di.fc.ul.pt
WWW home page: http://www.navigators.di.fc.ul.pt/

Abstract. When designing communication protocols there is always a tradeoff between generality and performance. This chapter reports one approach to achieve right balance between these two aspects, using a network model that 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. The approach consists in making visible relevant *topological* aspects of the uderlying network infrastructure to the protocol designer, and is illustrated by several algorithms that use topology information to achieve improved performance.

1 Introduction

When designing communication protocols there is always a tradeoff between generality and performance. Generic approaches make few assumptions about the underlying network. The resulting protocols are easy to port to different network structures but often exhibit poor performance. On the other hand, tailored solutions exploit particular features of a given class of networks in order to achieve better performance. However, tailoring has its disadvantages: if the features exploited are peculiar only to one or two networks, it may be difficult, if not impossible, to port the resulting protocols to other networks that do not own these characteristics. The successful design must capture the right balance between generality and performance.

This chapter reports one approach to achieve this balance, using a network model that 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. This model, that we simply call "WAN-of-LANs", was central to the design of the NavTech architecture. We are going to present a number of innovative ways to take advantage of it, by making visible relevant *topological* aspects to the protocol designer.

* This report has been published as a chaper of the book "Topology-aware algorithms for large-scale communication. *Recent Advances in Distributed Systems*, S. Krakowiak and S.K. Shrivastava (editors), Captulo 6. Springer Verlag LNCS vol. 1752."

The chapter is organized as follows. Section 2 presents the NavTech framework and network model on which our protocols are based. The next three sections describe protocols exploiting that communication infrastructure. The first protocol, presented in Section 3, is a clock synchronization protocol that is based on an hierarchical composition of a protocol tailored to broadcast LANs with a protocol that makes use of the GPS architecture. The second protocol, presented in Section 4, 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. The third protocol, presented in Section 5, is a total order protocol that dynamically adjusts the ordering algorithm as a function of the participant location in the network topology. Related work is discussed in Section 6 and Section 7 concludes the chapter.

2 The NavTech Framework

This section describes the main characteristics of an architectural framework called NavTech. The framework specifies architectural constructs and mechanisms to assist the design and execution of dependable distributed applications in large-scale settings. Some of these constructs were the enabling factors of topological-awareness as we describe it in this chapter. In other words, the topology-aware protocols that we will encounter in the following sections were designed having in mind the availability of an infrastructure with the characteristics described here. That is, NavTech is essentially a macroscopic framework made of components such as the Internet, private local area 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 infra-structural networking and computing technologies, and in that sense it is an open architecture, applicable to systems built from COTS components.

2.1 The Scale Problem

NavTech was designed for large-scale applications. Topology issues become more important in the measure that distributed systems grow in span, complexity and number of sites. In order to take advantage from topology, we must identify the implications of scale on the structure of distributed systems, under their several facets: the computation participants; the communication system; the interaction styles.

Computation Participants Scale affects computations in several ways. The most obvious aspect of scale concerns the number of participants in the computation. The number of entities simultaneously involved in a computation varies, according to the type of interaction concerned, but that number is often significantly smaller than the number of sites in the network. For the sake of simplifying our forthcoming analysis, we propose to consider a coarse-grain scale metric, of

three "levels": very-large-scale— order of millions and up; large-scale— order of thousands up to the million; small-scale— order of hundreds down.

The communication system characteristics have a fundamental impact on the scale of computations, since they make it extremely difficult, and sometimes even impossible, to reproduce in large-scale the operating conditions that are otherwise found in small-scale systems. In consequence, there is a need for structuring applications in ways that allow reasonably performant operation. Hierarchical organization and clustering according to the topology of the infrastructure, are paradigms addressing this particular issue in NavTech.

Communication System A large-scale network such as the Internet forms what we might call the *global network*, for the purpose of large-scale computing. A number of sites in the order of 10^7, and growing, puts it in the very-large-scale level, with a number of *structural* characteristics dictated by its scale and technology: sparse connectivity; limited diffusion capabilities; weak reliability and timeliness; globe-wide distances; public-domain or standard protocols.

In face of these characteristics, we can extract a set of functional communication properties, the most important of which are listed below, deriving both from sheer scale and from technology shortcomings: large communication delay variance; asynchronism; partitioning (e.g. set $\mathcal{M}$ of sites reach each other and set $\mathcal{N}$ of sites reach each other, neither reaches the other set); non-transitivity (e.g. A reaches B, B reaches C, but A cannot reach C); non-symmetry (e.g. A reaches B, but B cannot reach A).

On the other hand, inside what we might call *local networks*, the infrastructure takes significantly different characteristics that should not be ignored: availability of LAN or MAN technology (including the foreseen role of ATM); dense connectivity (normally broadcast-level); good reliability and timeliness; private operation, enterprise-oriented. Such significant differences should not be ignored by a large-scale architecture. A moderate number of sites puts local networks in the small- to large-scale level.

This analysis identifies a fundamental topological paradigm: many networks exhibit physical organization as a 2-tier WAN-of-LANs.

2.2 Networking Model

The networking model of NavTech is 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.

2-tier WAN-of-LANs The 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

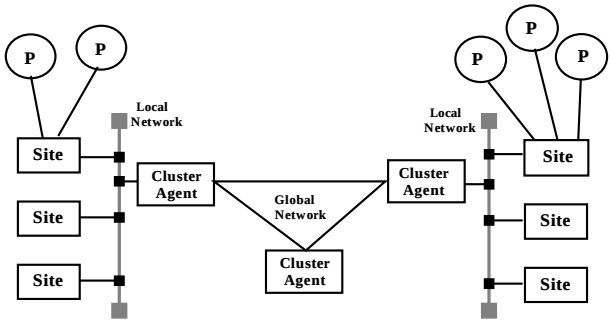


Fig. 1. 2-tier WAN-of-LANs and clustering as followed in NavTech

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.

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 *Facility* as a cluster of sites and the *Site* as a cluster of participants.

The first clustering level is obviously compatible with the 2-tier architecture identified in the previous section, and is illustrated in figure 1. Clustering sites that coexist in a same local network can simplify internetwork addressing, communication and administration of these sites. These sites are hidden behind a single entry-point, a *cluster-agent*, a logical gateway that represents the local network members, for the global network. Organization-dependent clustering allows to run specific protocols behind the cluster-agents, without that colliding with the need to use standard protocols in wide-area networking. Global network communication is then performed essentially among cluster-agents.

The second level of clustering consists in taking advantage of a multiplying factor between the number of sites and the (sometimes large) number of participants that are active in communication.

This distinction between *sites* and *participants* is in favor of – and can only be achieved by – a *communication subsystem* approach for structuring the machine's networking. A site-level *protocol server* should take care of all send and receive activities on behalf of the participants residing locally to it.

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

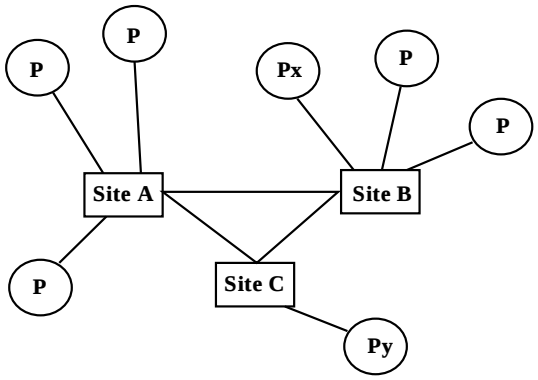


Fig. 2. Site-participant duality

Sites and Participants The NAVTECH platform supports interactions among entities in different sites (e.g. processes, tasks, etc.). For NAVTECH, these entities, that can be senders or recipients of information, or both, are *participants*. Participants interact via *sites*, which handle all communication aspects on behalf of them, as represented in figure 2. A system built to the site-participant duality provides a framework for defining domains of different synchrony and reliability. For example, intra-site communication is easily made synchronous and reliable, reducing the asynchrony and unreliability problem to inter-site communication. In consequence, while site failure detection is unreliable [4], participant failures can be reliably detected.

Hierarchical network model Following the structuring principles outlined above, we can derive a network model that has the following components, as illustrated in Figure 3. Protocol participants are simply designated processes and have disjoint memory spaces. The system is composed of a collection of processes $\mathcal{P} = \{p_1, \ldots, p_n\}$. We assume that a unique identifier is associated with each process $p \in \mathcal{P}$ (for convenience, we will use the same notation to refer to the process and its identification). We also assume that an order relation $\prec$ can be defined between process identifiers.

Processes are executed in *sites* or *nodes* (we use both terms interchangeably in the discussion). When two processes reside on the same node, messages on to and from these processes to the rest of the network can be filtered by a process on behalf of the operating system (usually, by the kernel itself). The mapping between processes and nodes can be also used for topology-aware failure detection using simple decision rules. For instance, when a node crashes, all processes running on that node are forced to crash.

Nodes are interconnected by two broad classes of *networks*: local networks (for example, using LAN or ATM technologies) and long-haul links. Local-area networks are assumed to have high-bandwidth, low latency and high reliability. Local networks are also less prone to network partitioning than long-haul links.

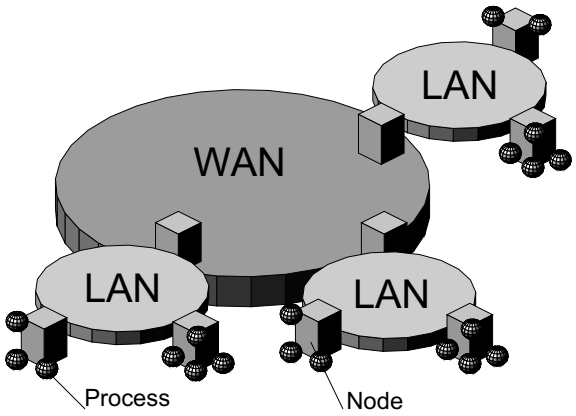


Fig. 3. Communication architecture.

Long-haul links from the the interconnecting mesh that supports communication among local-area clusters. They have lower bandwidth, exhibit higher latency and are prone to partitioning. Some nodes, the cluster-agents, assume the role of gateways, forwarding messages from and to the local networks to the long-haul links. Cluster-agents can be made visible when convenient, as for instance in the causal order protocol.

2.3 The Architecture of NavTech

The overall architecture of NavTech is represented in figure 4. The NavTech platform lies on an abstraction of a multipoint network, the *Abstract Network, AN*, created over the physical infrastructure. The *Site Failure Suspector, FS*, is in charge of assessing the connectivity and liveness of sites, and depends on the Abstract Network, by listening to traffic going into each site, and by exchanging information with other FS. The *Site Membership, SM*, depends on information given by the FS module. Based on the latter it creates and modifies the membership of site-groups. The *Communication Support Services*, *CS*, implement group communication and clock synchronization. All the four modules described are topology sensitive. Protocols may run differently depending on whether: they run in a local network; they run in the global network; this node is a cluster agent. Together, they also form the 'site' part of a node.

The *Participant Membership* module *PM* creates and modifies the membership of participant groups, and validates activity conditions, such as *majority* in a primary partition. The *Participant Failure Detector* module, *FD*, is a module with strictly local operation. Based on probes implemented using available O.S. support, it assesses liveness of participants. The *Activity Support Services* module *AS* implements protocols and algorithms that assist participant activity, such as replication management, mutual exclusion, cooperation awareness, etc. These three modules form the 'participant' part of the node, materializing

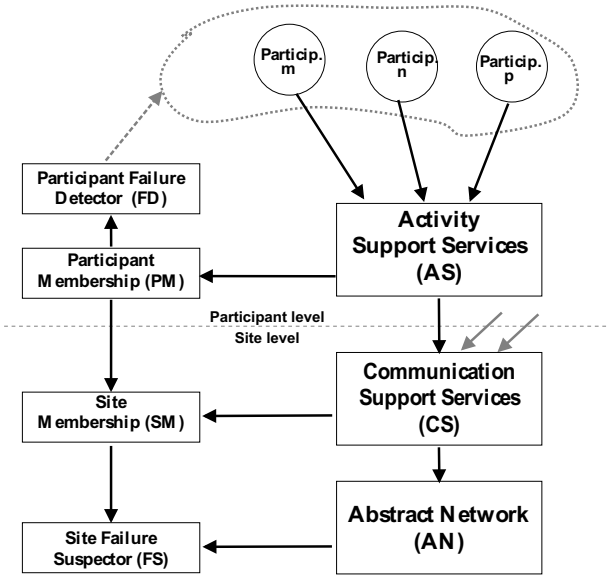


Fig. 4. NavTech architecture: 'depends-on' relation

the site-participant hierarchy. There can be several of these modules mapped onto the site part, as suggested by the grey arrows. NavTech uses groups as its central paradigm. However, participant groups are concerned with distributed applications, and delegate 'communication' on the site group that represents them, which runs the communication protocols. Normally, a site group SG_m is composed of the nodes that host members of participant group G_m. Additionally, there can be several groups of participants mapped onto one site group.

Related transport protocols In order to preserve the maximum generality possible, the hierarchical nature of our network model is only made visible when necessary, for example when important performance gains can be achieved. Thus, most of the design follows quite generic assumptions about the companion communication protocols required to support our topology-aware services. Since not all of the protocols we are going to describe rely on the same underlying communication services, we clarify this issue here.

The causal protocol simply assumes that the underlying message passing subsystem is reliable, in the sense that messages sent by a correct process are always received by all correct addressed participants. This assumption is consistent with recent implementations of other causal multicast protocols, which are based on a reliable transport layer (for instance, HORUS [29]). For the causal protocol we do not need to make any assumptions about the order in which messages are received. The identification of the sender $s_m \in \mathcal{P}$ and the set of destination processes $\mathcal{A}_m \subseteq \mathcal{P}$ are always associated with each message m. We also do not impose any restriction on the destination addresses: a message can

always be sent to any set of processes in $\mathcal{P}$. Additionally, we assume that each sender assigns a locally generated integer value c_m, to each message, such that if m is sent before n then $c_m < c_n$ (this can be trivially obtained by using a local counter [2] c_p at each process p). Although the pair (s_m, c_m) uniquely identifies a message, for convenience we define the message's *unique identifier* also including the destination address, that is, $uid_m \equiv (s_m, c_m, \mathcal{A}_m)$.

On the other hand, the total order protocol makes a stronger assumption, requiring the availability of a reliable (unordered) multicast mechanism and its associated membership service (module SM)[3]. We further assume that the multicast mechanism follows the virtually-synchronous model as defined in [2] and, informally, guarantees that all messages are received by all group members. The only assumption about the order in which messages are received is that all logical point-to-point channels between any pair of processes are FIFO (this can be easily enforced using sequence numbers). The membership service is responsible for giving each process information, called *views*, about the operational processes in the system. It is assumed that views are *linearly* ordered $(V^i, V^{i+1}, ...)$, i.e., that in case of network partitioning only a majority partition remains active and continues to receive views.

As in most works in causal and total multicast protocols, in addition to the *send* event, we introduce two distinct events: *receive* and *deliver*. The receive event identifies the message arrival at the protocol layer and is not visible at the application level. The deliver event identifies the message arrival at the application level. In order to enforce a given ordering discipline, the protocols may have to delay the delivery of received messages.

Simulation model In order to measure the performance of some protocols we resorted to simulation. For that purpose, MIT LCS Advanced Network Architecture group's network simulator (NETSIM [10]) was used. The network delay $D_{(s,r)}$ between the sender s and the recipient r is represented by a probability distribution function, with a mean value of $\mu_{(s,r)}$, and a variance of $\sigma^2_{(s,r)}$. According to our assumptions about the underlying infrastructure, the distribution function depends on the type of network being used to interconnect each sender-recipient pair. Different network latencies were assigned to local-area network and long-haul links (actual values will be presented when describing the protocols). A node receives its own messages with negligible delay.

3 Topology-aware Clock-synchronization

The first example of an algorithm exploiting the WAN-of-LANs network model is a clock synchronization algorithm. Our global time service for large-scale (world-wide) systems based on a topology-aware approach is called CesiumSpray and

[2] Since this counter is stored in a bit array with limited capacity, we cannot have indefinitely growing counters. However, techniques exist to overcome this problem [15].

[3] A number of recent systems [3, 19, 29] also implement total order on top of reliable multicast services.

its principles were already introduced in a previous Chapter. In this section we highlight the topology-aware features of the algorithm.

3.1 Time Service Architecture

The topology-aware clock synchronization exploits the WAN-of-LANs network model in the following way:

- At the LAN level, a protocol tailored to local area networks is used. The protocol fully exploits the intrinsic attributes of these networks: error rate is low, transmission delay is bounded but with high variance, median transmission delay is close to the minimum, and message reception is *tight* in absence of errors, meaning that the low-level message reception signal occurs at approximately the same time in all nodes that receive it. This feature can be made fully deterministic when operating from real-time kernels. It is a crucial feature for the mechanism underlying the synchronization algorithm, as will be shown ahead.
- At the WAN level, the GPSs VavStar satellite system is used as the "global network" link between local networks.

The integrated solution combines the LAN-level algorithm with the WAN-level service in a hierarchical manner. CesiumSpray can be implemented on virtually any large-scale distributed computing infrastructure as we see them today— such as the wide-area point-to-point Internet. Given its hierarchical nature based on GPS NavStar, it has virtually unlimited scalability. It is particularly well-suited for large-scale real-time systems.

The architecture of CesiumSpray is shown in Figure 5. The clock synchronization scheme is a hybrid of external and internal synchronization. The top-level of the hierarchy is the source of absolute time, the NavStar GPS, which performs external synchronization by "spraying" its time over the set of GPS-nodes. The second level of the hierarchy is formed by every local network of the system, with the condition that each be provided with at least one GPS-node. The external time resident in the GPS-node is further "sprayed" inside the local network through an internal synchronization algorithm.

3.2 Advantages of Topology-awareness for Clock Synchronization

Current Internet-based synchronization schemes, such as NTP[17], effective as they may be today, cannot reach the effectiveness of CesiumSpray, because they do not relate the synchronization architecture to the network architecture. For example, the location of external time masters in NTP is not related to the existence of local networks with broadcast properties, such as in CesiumSpray: reading a master clock may mean crossing several Internet gateways; neighbor clocks in the same LAN do not take advantage of that fact.

The topology-aware approach presented here does not suffer these drawbacks. On the other hand, the performance of CesiumSpray synchronization in its current form depends on the assumed real-time capability of the kernels to bound the interrupt delay variance.

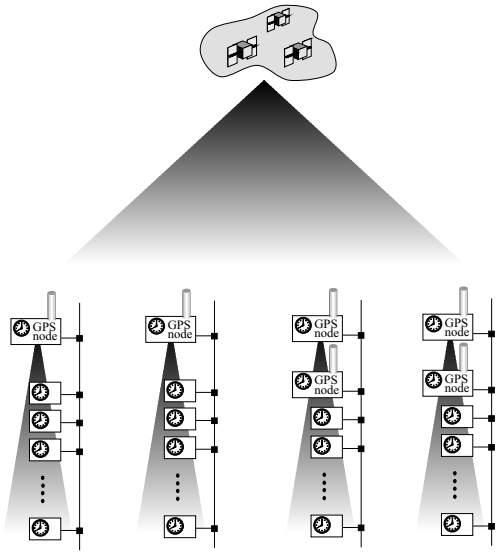


Fig. 5. The architecture of CesiumSpray

4 Topology-aware Causal Communication

The second example of a topology-aware algorithm, is a causal communication layer that delivers messages in causal order. This layer enforces a *logical precedence* [2]:

Logical precedence: *In a distributed system, in which information is exchanged only by transmitting messages, a message m is said to* precede *or to be potentially causally related to a message n, represented as $m \rightarrow n$, only if: (i) m and n were sent by the same process and n was sent after m or; (ii) m has been delivered to the emitter of n before n was sent or; (iii) there exists x such that $m \rightarrow x$ and $x \rightarrow n$.*

Experience has shown [2, 19, 29] that the design of distributed applications can be simplified if messages are received in order of logical precedence. Since extra complexity would be added to such applications, should the communication subsystem not provide causal delivery, several algorithms have been proposed to implement this ordering discipline [2, 19, 13, 20, 3]. Nevertheless, despite its advantages, the use of causal communication has been somehow limited by the overhead incurred by existing implementations. A major cost of protocols that preserve logical precedence is the size of "history" information that needs to be stored and exchanged to maintain causality, specially in large-scale systems where group addressing is used.

We now show how a topology-aware approach can benefit from the WAN-of-LANs model, allowing to extend previous results on causal history compression using knowledge on the topology of the communication structure. Our compression technique uses the concept of a *causal separator*, a set of nodes of the

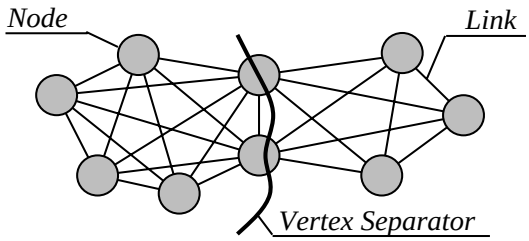


Fig. 6. Vertex separators

communication graph that can be used to filter causal information [23]. An implementation of this optimization is presented. Then we show the applicability of this approach to real-life large-scale networks with a hierarchical nature: we present a methodology to model the communication system as a graph, where causal separators that match the underlying physical and administrative organization are clearly identified.

4.1 Causal Separators

It has been shown that when communication graphs have a process that acts as a gateway, it is possible to decrease the amount of information required to determine temporal relationships [16]. Optimizations based on the communication patterns of processes have also been presented in [28]. Our work extends these results to arbitrary communication structures, making the following contributions:

- we show that, even in graphs that contain cycles, it is possible to reduce the size of the information exchanged by defining *causal separators*, a set of nodes of the communication graph that can be used to filter causal information.
- we present a methodology to model the communication system such that one can make practical use of the previous result.

These two aspects will be dealt with considerable detail in the next two sections. We first provide a global overview of our approach.

We assume that each process is able to communicate directly only with a given subset of system processes $\mathcal{L}_p \subseteq \mathcal{P}$. Two processes not directly linked can communicate indirectly through intermediate hops (usually, automatically selected by a routing algorithm). The complete communication topology can thus be represented by a graph $G(\mathcal{P}, E)$, where processes are the vertices and the communication links between them the edges: there is an edge incident to $\{p, q\}$ if p can send messages to q. We assume that the graph is connected. A set of processes, S, is called a (F_S, B_S) vertex separator, where the sets F_S and B_S are called, respectively, *forward* and *backward* sets, iff $F_S \cap B_S = \emptyset \wedge F_S \cup B_S \cup S = \mathcal{P}$ and $\forall f \in F_S, \forall b \in B_S$ every path connecting f and b passes through at least

one vertex of S. In the context of causal communication, we called such vertex separators *causal separators* (see Figure. 6).

In the next sections we will show that a causal separator can work as a barrier that filters all information concerned with messages exclusively addressed to the backward set and "reported" to all members of the separator ("reported" will be defined precisely). Let a set of processes bounded by one or more causal separators be defined as a *causal zone*. An interesting corollary of the previous observation is that the causal information concerned with the elements of a causal zone may never need to be propagated outside the boundaries of that causal zone, if *reported* to all members of the separator from which a causal relation with the outside is established.

In Section 4.3 we will show that it is possible to exploit the WAN-of-LANs model in such a way that causal separators can be effectively used. The technique exploits the physical structure of existing networks, in particular its hierarchical nature, to create a communication graph where causal separators match the underlying physical and administrative organization.

4.2 Causal Order Algorithm

In this section we present an implementation of causal histories that allows to reduce the causal information exchanged using information about causal separators. The interested reader will notice that there are no deep fundamental differences between our representation of causal histories and alternative approaches described in the literature [2, 19]. However, it provides the ground for the implementation of optimizations based on causal separators, the main contribution of our work. Our representation of the causal history, that we called *extended causal history*, stores causal information in three different entities:

- a *causal history*, $\mathcal{H}_p$, a list with the messages that precede the next message to be sent by p;
- the *delivery history*, $\mathcal{D}_p$, a list with the messages that have already been delivered at p and;
- a *carbon-copy* history, $\mathcal{C}_p$, that keeps track of to where causal information has been "reported" (the carbon-copy history is used for compression of causal information, and its use will be detailed later).

The delivery history maintains a record of all messages that were delivered to a given process. The causal history maintains a record of all messages that precede the next message to be sent by a given process. Although the causal history contains the delivery history, different compression rules will be later applied to each, thus we decided to explicitly maintain this information in two different entities (explicit separation between causal and delivery histories is also used in other approaches, for instance [20]). These histories are used in the following way:

Every time a message m is sent by process p, it is timestamped with its sender's causal history, $\mathcal{H}_p$. All messages in $\mathcal{H}_p$ are then said to be "*reported*" to

all recipients of m. This information is kept in carbon-copy history, $\mathcal{C}_p$. When a message is received by process q, the recipient compares the message timestamp with its own delivery history and checks whether or not all preceding messages have been already delivered locally. It then delivers or delays the received message accordingly. When a message is delivered, the recipient delivery and causal histories are updated accordingly.

More precisely, causal delivery can be enforced using the delivery and causal histories if the following rules are applied (for clarity, we will defer the use of the carbon-copy history until rule 6 is introduced):

R1 (Initial state): When p starts execution, $\mathcal{H}_p$, $\mathcal{D}_p$, and $\mathcal{C}_p$ are empty. Also, $c_p = 0$. □

R2 (Timestamping): Before being sent by process p, a new uid is assigned to message m by incrementing the local counter c_p. Next, m is timestamped with p's causal history, that is, $\mathcal{H}_m = \mathcal{H}_p$. □

R3 (Causal delivery): On receipt of message m sent by process p and timestamped with a causal history $\mathcal{H}_m$, process $q \in \mathcal{A}_m$ delays the delivery of m until all messages in $\mathcal{H}_m$ that were addressed to q have been delivered at q [4]. More precisely, q delays the delivery of m until the following condition is true: $\forall (i \in \mathcal{H}_m : q \in \mathcal{A}_i) \quad i \; \textit{is-in} \; \mathcal{D}_q$; where the *is-in* relation is here defined as: $m \; \textit{is-in} \; \mathcal{D} \iff m \in \mathcal{D}$. □

R4 (Record maintenance): When process p sends m it atomically adds m to $\mathcal{H}_p$ and to $\mathcal{D}_q$. When a message, m, is delivered at $q \neq s_m$, m's timestamp, $\mathcal{H}_m$, is added to $\mathcal{H}_q$. Additionally, m is added to $\mathcal{H}_q$ and $\mathcal{D}_q$. □

Rules 1-4 above are enough to enforce causal delivery of messages (see [23] for a proof). However, this solution suffers from a serious drawback as, unless some measures are taken to garbage-collect redundant elements, the causal histories continue to grow indefinitely. In the next paragraphs we present an extra set of rules that allow the garbage-collection of the extended causal history.

We start by compressing the delivery history. The compression rule exploits the fact that messages from the same sender must always be delivered in the order they were sent. From the causal delivery rule, if a message m from process p is delivered to q, then all previous messages from p, addressed to q, were already delivered at q. As a result of this rule, delivery histories do not need to keep more than one message from each sender. More precisely,

R5 (FIFO Delivery): At most one unique message identifier needs to be stored from each sender in the delivery history. When a message m from process p is added to $\mathcal{D}_q$, m replaces the previous message from p delivered at q.

Naturally, since some elements are deleted from the delivery history as new members are added, the definition of "is in $\mathcal{D}$" must be slightly changed. We now

[4] A message can always be delivered without delays to its sender.

say that a message m *is-in* $\mathcal{D}$ if and only if there exists a message in $\mathcal{D}$, from the same sender, with a higher or equal identifier. More precisely, m *is-in* $\mathcal{D} \iff \exists_{n \in \mathcal{D}} : c_m \leq c_n$. □

We now garbage-collect the causal history. The idea is to remove from this history all the elements not strictly required to preserve causal delivery. In doing so, we discard information about the past. The method is an extension of the "last send" and "last update" vectors proposed by Singhal and Kshemkalyani [27], also suggested in [28] to optimize the use of vector clocks on overlapping groups. We simply extend this method to arbitrary addressing schemes. The optimization can be informally presented as follows: before being sent, message m is timestamped with its sender's causal history $\mathcal{H}_p$. It will then be delivered to $\mathcal{A}_m$, after all messages in $\mathcal{H}_m$. Any message $n : \mathcal{A}_n \subseteq \mathcal{A}_m$ that carries m in its timestamp, does not need to carry $\mathcal{H}_m$ since it will be delivered after m, thus after all messages in $\mathcal{H}_m$. However, this requires some bookkeeping of whom the messages were reported to. This information can be kept in an additional history, called the *carbon-copy* history, $\mathcal{C}$. The carbon-copy history contains a field for each message in the causal history, storing the list of processes to which the associated message was already "reported" within the timestamp of another message. The carbon-copy history should be updated using the following rule:

R6 (Carbon-copy): Each process, p, keeps a carbon-copy history, $\mathcal{C}_p$, that contains an element $\mathcal{C}_p(m)$ for each message m in $\mathcal{H}_p$. These elements are used according to the following rules:

Extended timestamping (optional): When a message m is timestamped, in addition to $\mathcal{H}_p$, it may also by timestamped with $\mathcal{C}_p$. We refer to the carbon-copy field of m's timestamp as $\mathcal{C}_m$.

Send update: After sending a message m, and before inserting m in $\mathcal{H}_p$, update all fields of $\mathcal{C}_p$ as follows: $\forall (i \in \mathcal{H}_p)$ let $\mathcal{C}_p(i) = \mathcal{C}_p(i) \cup \mathcal{A}_m \cup \{s_m\}$. Then insert m in $\mathcal{H}_p$ and initialize $\mathcal{C}_p(m) = \emptyset$. These updates should be performed in a single atomic operation.

Deliver update: After delivering a message m, processor $q \neq s_m$ updates the carbon-copy fields of previous messages from the same sender as follows:

$$\forall (n \in \mathcal{H}_q : s_n = s_m \wedge c_n < c_m) \text{ let } \mathcal{C}_q(n) = \mathcal{C}_q(n) \cup \mathcal{A}_m$$

Then, it adds all elements n of $\mathcal{H}_m$ to $\mathcal{H}_q$. The carbon-copy fields of these messages are initialized as follows (if extended timestamping is not used, use $\mathcal{C}_m(n) = \emptyset$):

$$\mathcal{C}_q(n) = \mathcal{C}_m(n) \cup \mathcal{A}_m \cup \{s_m\} \cup \bigcup_{i \in \mathcal{H}_q : s_i = s_n \wedge c_i > c_n} \mathcal{A}_i$$

If $n \in \mathcal{H}_m$ is already in $\mathcal{H}_q$ it just updates the existing carbon-copy field, merging $\mathcal{C}_q(n)$ with the result of the previous expression. Finally, q inserts m in $\mathcal{H}_q$ and initialize $\mathcal{C}_q(m) = \{s_m, q\}$. These updates should be performed in a single atomic operation. □

The carbon-copy history is used to compress the causal histories in the following way: (1) messages do not have to be included in a timestamp, if they have already been included in a timestamp of another message sent to the same destination; and (2) when the carbon-copy field of a message completely includes the message's address, that message can be safely removed from the causal history as it has already been reported to all relevant processes. More precisely,

R7 (Timestamp Redundancy): When timestamping a message m, processor p only includes in $\mathcal{H}_m$ the elements of its causal history $i \in \mathcal{H}_p$ not reported to $\mathcal{A}_m$, according to p's knowledge, i.e.: $\mathcal{H}_m = \bigcup i \in \mathcal{H}_p : \mathcal{A}_m \not\subseteq \mathcal{C}_p(i)$.

R8 (History Redundancy): In a causal history, $\mathcal{H}_p$, if there exists a message, m, such that $\mathcal{A}_m \subseteq \mathcal{C}_p(m)$, m can be removed from $\mathcal{H}_p$ and $\mathcal{C}_p$.

4.3 Using the Communication Topology

Causal separators can be exploited to reduce the size of message timestamps as follows. When a member of the causal separator timestamps a message addressed to processes exclusively located in the forward set, it can omit in the timestamp all elements of its causal history that were addressed exclusively to members of the backward set and that were already reported to the other members of the causal separator. More precisely,

R9 (Topological timestamp): Processor p is sending a message m. All messages $n \in \mathcal{H}_p$ for which exists a (F_S, B_S) causal separator[5], S, such that: $p \in S \wedge \mathcal{A}_m \subseteq F_S \wedge \mathcal{A}_n \subseteq B_S \wedge S \subseteq \mathcal{C}_p(n)$, do not need to be inserted in $\mathcal{H}_m$

The compression achieved with the topological timestamping rule can be further improved at the cost of reporting causal information to all the members of the causal separator. In fact, remember that the carbon-copy fields can always be forced to a given desired value just by sending a message to the relevant processes. For proofs of correctness of our rules see [23].

There are a number of challenges associated with the use of our topological timestamping scheme. Firstly, any change to the topology can alter the membership of the causal separators: solutions for this problem are discussed in [23]. Secondly, arbitrary network can have a large number of causal separators: topological timestamping can be applied to all separators or just to a subset of them. Thirdly, causal separators need to be computed before the topological timestamping rule can be applied. Several algorithms to identify vertex separators in a graph are available (for instance, see [8]), but these can be too expensive to be executed frequently during normal system operation.

Thus, our method is better suited for applications where the topology is relatively static or can be computed at compile or configuration time. In this case causal separators may be computed in advance, and the corresponding forward

[5] Where F_S and B_S are respectively the forward and backward sets of the separator(see chapter 4.1)

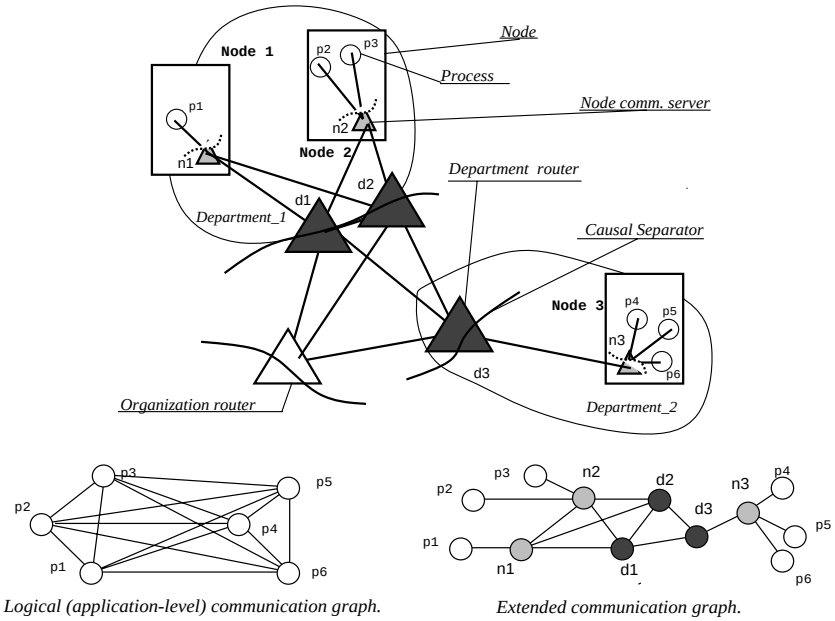


Fig. 7. The communication structure and the communication graph

and backward sets prepared and loaded in all causal separator members to allow a fast execution of the topological timestamping rule. A particular case of a relatively static topology, is the one defined by the network infrastructure that connects individual nodes of a distributed system. In particular, the WAN-of-LANs model offers an excellent ground to define causal separators in a meaningful way. We propose a solution for the provision of causal communication in large-scale systems based on the following methodology, that consists in mapping protocol entities onto architectural components:

- the communication entity that connects a node to the network assumes the role of a router process in the communication structure (the site representative in NavTech). This process is a causal separator that connects the machine to the network.
- routing of messages between nodes is also implemented by special processes, usually placed in dedicated router nodes (the NavTech cluster agent), able to execute topological timestamping as they forward the messages.

Using this methodology, one can build an *extended communication graph* that takes into consideration the organization of the underlying infrastructure. In this graph, new nodes are added, corresponding to communication server and router processes (see an example in Figure 7). Promoting the communication entities to nodes of the (extended) communication graph has several advantages:

- It provides a useful way of mapping the abstract communication graph onto the existing physical and administrative communication structure. This gives the means for the practical and useful identification of causal separators (for instance, node's communication server, department router, etc.).
- The physical communication structure (i.e., the servers and routers, not the application processes) is usually relatively static. This feature provides the basis for an efficient exploitation of topological timestamping and makes the costs associated with dynamic changes of the structure almost negligible.
- It provides a practical way of exploiting communication locality. Using topological timestamping on the structure obtained by adding communication and router processes, it is possible to prevent local information from being propagated on the network. Messages exchanged between processes of the same node are filtered out by the node router, messages exchanged between processes in the same department by the department router, and so on.

In conclusion, we believe that the physical and administrative organization of the physical communication structure can be exploited to support the efficient implementation of topological timestamping. In order to achieve this goal, some communication elements must be promoted to nodes of the communication graph.

4.4 Performance of Causal Separators

In order to evaluate the performance of our approach we resorted to simulation. We measured the average size of timestamps obtained using our extended causal histories and compared these values with the size required by a non-optimized clock-matrix approach [20].

From the several simulation results presented in [23], we have selected a configuration of three LANs connected in such a way that circles exist both at the logical and physical level, as illustrated in Figure 8. If the communication infrastructure is constructed in such a way that, in each broadcast network, both gateways receive all out-going traffic (since these are connected to a broadcast network this can be achieved with negligible cost), the gateways can act as causal separators and filter causal associated with local communication.

The impact of applying topological timestamping is this case is illustrated in figure 8. The figure shows the distribution of message timestamp size in group $G1$ with and without separators. The figure shows also the distribution of the host history size on process $p1$ with and without separators. It can be seen that, without topological timestamp, the causal history of process $p1$ is much larger.

4.5 Advantages of Topology-awareness for Causal Ordering

Early implementations of logical precedence (also described as the "happened before" relation) were based on logical clocks [14], a technique that introduces a systematic delay in message delivery and that orders more messages than those potentially causally related [26]. To avoid the disadvantages of logical

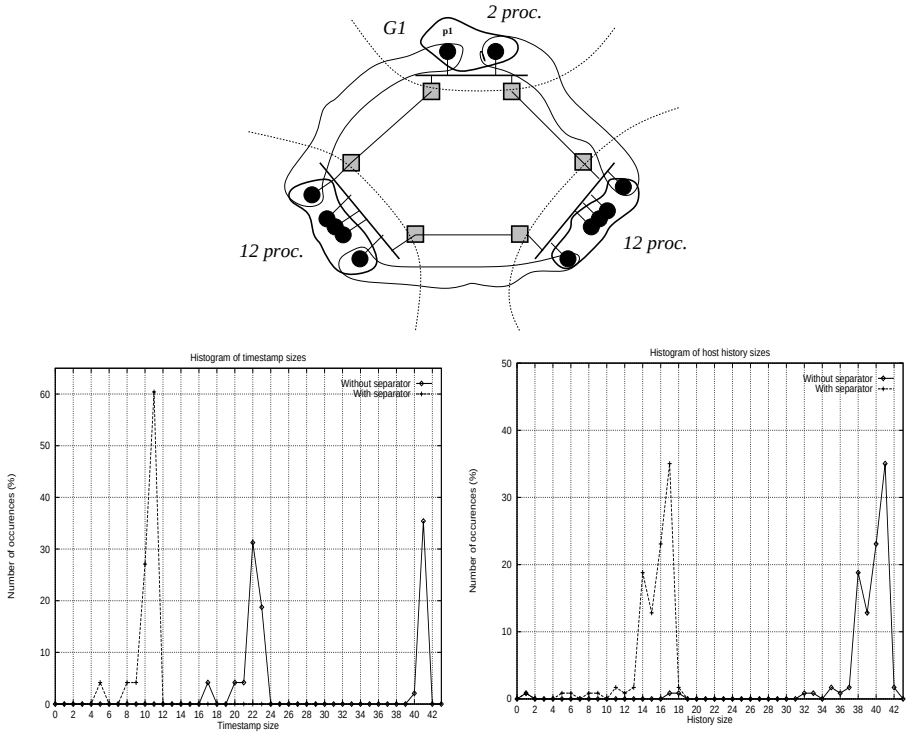


Fig. 8. Three interconnected broadcast networks and impact on group $G1$.

clocks, a new set of algorithms has been proposed, where the information required to precisely define causal relations is piggybacked on the messages exchanged. These approaches are based on causal histories or vector clocks [2, 19, 24, 13, 3]. Recent system have extended these approaches to systems that allow messages to be addressed exclusively to a subset of the existing processes [2, 13, 24, 20]. These early approaches do not take the topology into account and the amount of information that needs to be maintained and exchanged grows quadratically with the number of processes [6].

Our technique combines optimizations previously proposed by others [27, 28] with the innovative causal separator concept that allows to fully exploit the WAN-of-LANs model. We exploit the physical structure of existing networks, in particular its hierarchical nature, to create a communication graph where causal separators match the underlying physical and administrative organization. This approach can be applied to existing large-scale systems, fitting the WAN-of-LANs model, providing the means for using topological timestamping with little practical overhead.

5 Topology-aware Total Order

The third example of a topology-aware protocol is a total order protocol. Totally ordered multicast protocols have proved to be extremely useful in supporting many fault-tolerant distributed applications. For instance, total delivery order is a requirement for the implementation of replicated state-machines [25], which is a general paradigm for implementing fault-tolerant distributed applications.

Although several protocols have been described in the literature [1–3, 5, 7, 12, 13, 18], few were specifically targeted to operate in (geographically) large-scale systems. In a large scale network processes' traffic patterns are usually heterogeneous. The same applies to the network links: some processes will be located within the same local area network whereas others will be connected through slow links, and thus subject to long delays. In such an environment, none of the previous approaches can provide optimal performance.

The topology-aware total order protocol recognizes that some ordering mechanisms are more appropriate for local-area networks and other more suitable for the wide area network. Since we are targeting a WAN-of-LANs network model, we have designed an hybrid scheme, where each process is able to operate with the ordering mechanism that is most suitable given its position with regard to its peers in the network topology. If all processes are in the same cluster (single or set of interconnected LANs), one mechanism is used. If all processes are in different clusters, another mechanism is used. In intermediate scenarios, different mechanisms are integrated in a hybrid protocol [21].

5.1 Ordering Mechanisms

Among the several algorithms for implementing total ordering, the *token-site* [5, 12] and *symmetric* [19, 7] are the most used approaches. Both methods have advantages and disadvantages.

In the token-site approach one (or more) sites is responsible for ordering messages on behalf of the other processes in the system. This process works as a sequencer of all messages and is often called the *token* site. Token protocols are appealing because they are relatively simple and provide good performance when message transit delays are small (they are particularly well suited for local area networks). However, in a token protocol, a message sent by a process that does not hold the token experiences a delivery latency close to $2D$, where D is the message transit delay between two system processes (i.e., the time to disseminate the message plus the time to obtain either the token or an order number from the token holder). Thus, token-site approaches are inefficient in face of large network delays.

In the symmetric approach, ordering is established by all processes in a decentralized way, using information about message stability. This approach usually relies on *logical clocks* [14] or *vector clocks* [2, 19, 13]: messages are delivered according to their partial order and concurrent messages are totally ordered using some deterministic algorithm. Symmetric protocols have the potential for providing low latency in message delivery when all processes are producing messages.

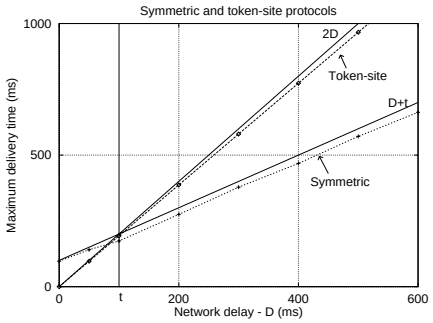


Fig. 9. Latency of symmetric and token-site protocols in isolation.

In fact, using a technique called rate-synchronization [21], symmetric protocols can exhibit a latency close to $D+t$, where t is the largest inter-message transmission time. Unfortunately, this also means that all (or at least a majority [7, 18]) of processes must send messages at a high rate to achieve low protocol latency.

In order to illustrate the behavior of both protocols in isolation we have selected the rate-synchronized symmetric protocol of [21] and the non fault-tolerant version of [12] (where a single site issues tickets on behalf of all other processes in the group). The measured latency of these protocols for different network delays is depicted in Figure 9, corresponding to a scenario where 30 processes send messages at a rate of 10 msg/s. It can be seen that, as noted before, the latency of the token-site protocol follows the $2D$ line and the latency of the symmetric protocol follows the $D+t$ line. The figure clearly shows that token-site protocols are more favorable when $2D < D + t \equiv D < t$, and that symmetric protocols are more favorable otherwise.

5.2 Static Hybrid Protocol

We now present a hybrid protocol for static topologies, i.e., topologies where traffic patterns, rates and communication delays are known *a priori* and do not change over time. The protocol is extended later to dynamic topologies. The hybrid protocol allows some processes to operate in symmetric mode (these processes are said to be *active*), or also called *sequencers*), and other processes to operate in token-site mode (these processes are said to be *passive*). At a given instant, each passive process is associated with a single active process which issues tickets on its behalf.

The protocol works as follows. Each process has a unique identifier p_i and keeps an increasing sent message counter c_i. Thus, each message is uniquely identified by the pair (p_i, c_i). Messages are multicast, using a virtually-synchronous primitive, directly to all processes of the group. Active processes keep an extra counter: the *ticket number* t_i. Ticket numbers are updated as according to the rate-synchronized symmetric protocol. A ticket is a triplet $(p_i, t_i, (p_j, c_j))$. An active process issues tickets for its own messages and for messages from its associated passive processes. At a given time, each passive process is associated with

```
forall process n set n mode to passive
let a be the process with highest rate; set a mode to active; set changed to true
while (changed is true) do // iteration
    set changed to false
    forall j such that j mode is passive do
        let a be the active process closest to j
        if (D(j,a) + t_j < 2D(j,a)) then
            set j mode to active; set changed to true
        else
            set sequencer of j to a
```

Fig. 10. Mode Assignment Heuristic

a single active process, called the passive process *sequencer* (an active process can be a sequencer of more than one passive process). Passive processes multicast their messages to all group processes which then wait for a ticket stating the total order of each message. The ticket is sent by each passive process's *sequencer*. In order to be disseminated to all processes, tickets are piggy-backed in messages sent by active processes. Tickets are ordered as in a symmetric protocol i.e., by increasing order of their ticket numbers, and tickets with the same ticket number are ordered according to the total order of ticket issuers. Finally, messages are delivered by the order of their associated tickets.

The critical part of the hybrid approach is to assign roles to each process. The decision must take into account the rate at which each process is producing messages and network delays between processes. In order to configure the system, a heuristic that analyzes each pair of processes in isolation is used. Consider a process n, subject to a load characterized by a mean inter-message transmission time t_n. Consider that the delay to the nearest (in terms of network delay) active process a is $D_{(n,a)}$. The condition that must be satisfied for process n to assume a passive role is $D_{(n,a)} + t_n > 2D_{(n,a)}$. In this case, inter-message transmission time is longer than the active process' round-trip delay and p can request and obtain a ticket from a before there is a new message to be sent. On the other hand, if $D_{(n,a)} + t_n \leq 2D_{(n,a)}$ since it is sending messages faster than the time required to obtain a ticket from the token-site, n should assume an active role (this not only offers lower latency but provides better load distribution).

The complete algorithm to assign roles can be obtained by applying the previous rule recursively, as described in Figure 10. Initially, all processes are made passive. Since at least one active process must exist in the system, the process (or one of the processes, if more than one exist) with smaller inter-message transmission time is selected as the initial active process. Then, the rule is applied to all other processes of the system to check if some of the processes should be promoted to active. This procedure is executed recursively until no change is made to the network.

5.3 Mode Switching Protocol

In order to apply the hybrid protocol to dynamic topologies, a protocol that allows a process to dynamically switch between active and passive mode is needed. This section describes such a protocol.

There are three types of transitions that can occur in a dynamic hybrid protocol, namely: (i) a passive process can change sequencer; (ii) a passive process can switch to active; (iii) and, an active process can switch to passive. Transitions can occur due to two main reasons: changes in the operational envelope and failures. Transitions due to failures happen when active processes, which are acting as sequencers of other processes, crash. In this case, passive processes associated with the failed sequencer must either select a new sequencer or become active. Transitions due to changes in the operational envelope happen when a process decides to adapt to new load or network delay conditions.

To guarantee correct operation, all active processes in the system must see the same sequence of configurations. Thus, the order in which transitions are executed, with regard to message flow and membership indications, must be agreed before transitions actually take place. In order to reach agreement about the $(i+1)$th configuration, the properties of the underlying view-synchronous layer (vs-layer) and the total order of messages, established by the ith configuration, are used. The vs-layer advantage is the guarantee that, in case of failures, all surviving processes receive the same set of messages before a new view is installed. This means that each view change is a synchronization point where all processes are guaranteed to have received the same messages. These properties greatly simplify switching protocols. However, no assumptions are made about the consistency of rates and network delay evaluations (i.e., no process can assume that some other process will change state just because such a transition is plausible according to its own local information): all transitions which are not directly triggered by failures must be initiated and disseminated by the switching process.

In order to describe the switching protocol some definitions are needed. Each process j is described by a triplet, called the *process descriptor*, denoted $D_j = (p_j, r_j, rn_j)$, where p_j is the process identifier, r_j is a role (one of *active* or *passive*), and rn_j is a *role-number* (role-numbers start with zero and are incremented every time a process changes roles). A *system configuration*, $C = \{V^i, \bigcup_{j \in V^i} D_j\}$, is defined as a system view plus the process descriptors of all processes in the view. It is also assumed that each process j keeps a record of the last of its own messages that has been delivered, l_j. Finally, it is assumed that a passive process p keeps the process descriptor of its sequencer in a variable called $S(p)$. The following text presents an informal description of the protocol.

Initial Configuration When the hybrid protocol starts, all processes must agree on some initial configuration. The exact configuration is not important since the system is able to reconfigure itself (as long as there is at least one active process in the initial configuration). An initial configuration was used

where all processes are active and remain in that state until they have received enough messages to evaluate traffic load and network delays.

Operation in Steady-state In the dynamic hybrid protocol, a process operating in passive mode is not statically assigned to a given sequencer process. Instead, a passive process can instruct any active process to order messages on its behalf, on a message-by-message basis (usually, a passive process only changes sequencer as a result of a configuration change). This confers flexibility to the system and provides fast adaptability to changes in the operational envelope. To allow dynamic binding, data messages are encapsulated in a protocol message with the following format: $\langle type, p_i, c_i, S_i, \textit{user-data} \rangle$, where p_i is the source, c_i is the message's sequence number and D_s is the process descriptor of the assigned sequencer for that message (the assigned sequencer will only issue a ticket if it still has the role-number specified in S_i when it receives the message).

Since messages can be transmitted concurrently with events that generate configuration changes, it is possible for the assigned sequencer to fail or increment its role-number before it has the opportunity to issue tickets for a group of messages. In order to cope with these cases, the protocol uses another special message, called a *reassign* message, with the following format: $\langle \textit{reassign}, p_i,]l_i, c_i], S_{\text{new}} \rangle$, where p_i is the source, $]l_i, c_i]$ is a range of message sequence numbers (the specification of this range will be described later on in this section) and S_{new} is the new sequencer for those messages. Reassign messages are only sent when the selected sequencer fails or becomes passive.

If a passive process fails, all of its messages, delivered by the vs-layer before the view change but not yet ordered by its sequencer, are silently discarded by all processes. This procedure is safe, because the properties of the vs-layer guarantee that tickets are totally ordered with respect to the view change.

Both active and passive processes store all received messages in a *pending* queue. Active processes issue tickets for their own messages and for all messages assigned to them in the pending queue (i.e., if the process descriptor in the message matches the process descriptor of that active process). Tickets are ordered as they are received (piggy-backed in the messages of the active processes). Finally, messages are removed from the pending queue and delivered by the order of their tickets. Although only active processes issue tickets, all processes (including passive processes) keep their ticket numbers synchronized according the protocol in [21]: a passive process may need to become active and the protocol exhibits better performance if these numbers are up to date.

Some messages are reserved for protocol usage and are not delivered to the user. The use of such messages will be clarified below. Also, the *reassign* message is never delivered: it is only used to update the sequencer field of all specified messages in the pending queue.

System Reconfiguration Process mode transitions and process crashes induce a sequence of system configurations. In order to voluntarily change their modes, processes broadcast special messages, namely $\langle\ \textit{goToActive}, p_i, c_i, S_i \rangle$

and $\langle$ *goToPassive*, $p_i, c_i, S_i\rangle$ messages. Such messages are sent in total order and their delivery triggers installation of a new system configuration. Processes may also be forced to change their mode due to failure of other processes thus, view changes also trigger installation of a new system configuration. Finally, passive processes may react to configuration changes by selecting a new sequencer. These situations will be addressed in the following paragraphs.

View changes Assume that the system is in configuration C^n is delivered by the vs-layer. A new configuration C^{n+1} is created. If there is no active process in such configuration (i.e, all active processes have failed) process m having the highest process unique identifier in V^{i+1}, is automatically switched to active mode (by setting r_m = *active* and incrementing rn_m); then, C^{n+1} is installed. If there is a passive process such that $S(p) \notin C^{n+1}$, this process selects a new sequencer m and sets $S(p) = (p_m, r_m, rn_m)$. Additionally, it sends message $\langle$*reassign*, $p_p,]l_p, c_p], S(p)\rangle$.

Transition from active to passive In the dynamic hybrid protocol there are two reasons for a process to change from active to passive: (a) its traffic load has decreased to a rate where it is more advantageous to request a ticket from another process; (b) or a nearby process has become active and transmitting at a much faster rate, so that there is no need to continue being an active process. In order to switch to passive mode, process p_i sends a special message $\langle goToPassive, p_i, c_i, S_i\rangle$, stops transmitting and stops issuing tickets, even for messages assigned to itself (these messages will eventually be assigned to another sequencer). It then waits for its own message to be delivered.

When the $\langle goToPassive\rangle$ message is delivered, before creating a new configuration, it should be checked if the sender is the last active process in the group. Note that since several processes can decide to become passive concurrently, all active processes might try to become passive but, since at least one active process must exist, the last one will fail. In the case the message is associated with the last active process, the transition is aborted (and the sender restarts sending messages and issuing tickets). Otherwise, a new configuration C^{n+1} is created by setting r_i = *passive*, and incrementing rn_i. Then, C^{n+1} is installed.

Transition from passive to active A transition from passive to active mode can happen either because a process becomes subject to higher traffic load, making the active mode a better choice, or because all active processes have failed and it is the process with highest identifier.

In the first case, passive process i broadcasts a special $\langle goToActive, p_i, c_i, S_i\rangle$ and stops sending messages. Then it waits until the special message is ordered by its sequencer and delivered. When the message is delivered, a new configuration C^{n+1} is created (by setting r_i = *active* and incrementing rn_i). Then, C^{n+1} is installed. All messages sent by i after this new configuration are ordered by process i itself.

In case of failure of the only active process, the passive process with highest identifier becomes active as soon as it receives failure indication from the vs-

layer. Upon this transition, new active process i issues tickets for all messages it has sent but that were not ordered in previous configurations, i.e., for all messages with sequence numbers in the interval $]l_i, c_i]$.

Change of sequencer A passive process can change its sequencer if some other process becomes active and the round-trip delay to that process is lower than to the previous sequencer. Since data messages specify the desired sequencer, sequencer switching is very simple. To avoid disturbances in FIFO ordering, passive process p stops transmitting temporarily and waits until all its previous messages have been delivered (i.e., it waits until $l_p = c_p$). It then sets its sequencer $S(p)$ to the new desired process descriptor and resumes message transmission.

A passive process p can also switch sequencer if its previous sequencer changes to passive mode. As before, the passive process sets its sequencer $S(p)$ to the new desired process descriptor. Additionally, knowing which message l_i was last ordered by the previous sequencer, it sends a reassign $\langle reassign, p_p,]l_p, c_p], S(p)\rangle$ message instructing the new sequencer to order unordered messages.

5.4 Dynamic Hybrid Protocol

Traffic patterns are likely to change over time in most interactive applications. Components usually react to incoming events by switching between idle periods and high activity periods. Networks delays are also subject to variations, due to load changes (office and night hours, for instance) or link failures (faster routes can become temporarily unavailable). Using the algorithms presented in the previous section, the dynamic hybrid protocol automatically adapts the operational mode of each process to changes both in traffic patterns and in communication links.

Evaluation of System Parameters To allow on-line reconfiguration, processes must be able to evaluate system parameters as traffic load and network delays. The following approach is used: each process timestamps every message with its own local clock at the time of transmission; based on the message's timestamp, all processes can determine the average transmission rate of the sender process. To determine delays in inter-process links, a simple round-trip delay method is used. At every pre-determined fixed interval of time[6], all receiving processes of a given data message respond immediately with a point-to-point null message to the originator process of the first message. This process can then calculate the delay between itself and all recipients. Note that the evaluation of link delays is also required by other components (for instance, for fault-detection), and can be implemented using low-level acknowledgments or synchronized clocks, for instance, using CesiumSpray.

In order to evaluate system parameters based on sample measurements, a simple *mean-shift* detector was used: an initial mean value of rate and delay is

[6] In our simulations, an interval of a few seconds was used.

calculated using the first k samples from each process[7]; whenever a run of k or more samples fall either all above the mean value or all below it, that mean value is recalculated and used in the next iteration. As the symmetric protocol relies on the fact that all processes must be constantly sending messages, system parameters can be evaluated after a short period of operation. This and other *mean-shift* detectors are described in detail in [11] and, as a future work, we plan to experiment with other detectors to evaluate their performance.

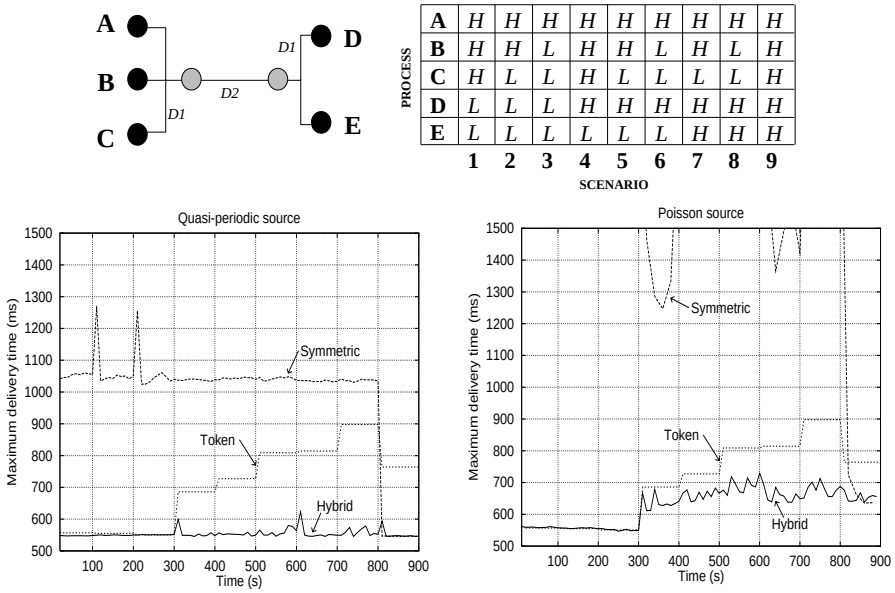
Switching Heuristic Section 5.2 showed how an external observer assign roles to each process in an hybrid configuration. Since an external observer can only be approximated, and to avoid centralized solutions, a heuristic that allows each process to make a local switching decision based on its own evaluation of the system state is now presented.

The heuristic is as follows: each process keeps track of its own message rate and of the network delay between itself and all other active processes. If its inter-message transmission rate is smaller than the delay to the closest active, it should switch to active mode, as shown previously. If, on the other hand, its inter-message transmission rate is higher than the delay to the closest active, it should switch to passive mode using the closest active process as its sequencer. To avoid frequent mode changes when the value of inter-message transmission rate is very close to the delay value, the decision to change mode is only made when the difference between these values becomes greater than a given threshold (a threshold of 20% of the delay value was used).

Performance of Total Order In order to test the effectiveness of our approach, the performance of the hybrid protocol is compared with that of the protocols selected in Section 5.1. The results were obtained with a system of five processes, connected by a network as shown in Figure 11 (grey nodes are network relays): processes A, B and C (and processes D and E) are within $D1$ of each other; however, the distance between a process in the first group and a process in the second group is $2D1 + D2$. Each process can be subjected to two different traffic loads, designated respectively by *high*(H) and *low*(L). Nine different scenarios were simulated, differing in relative traffic load of each process. The load of each process in each scenario is shown in the table (for instance, in scenario 1, process A has high load, process D has low load, and so on). In this simulation, the following parameters were used: $D1 = 20ms$, $D2 = 500ms$, a high rate of $100msg/s$ and a low rate of $1msg/s$ (both using a Quasi-Periodic source).

The performance results obtained using the heuristic above are also presented in Figure 11, alongside with results from symmetric and token-site protocols. The system ran continuously while the process load changed with time, making the system evolve through all nine scenarios in sequence. Results are presented for both a Quasi-Periodic and a Poisson message source. In the hybrid approach,

[7] In our simulations, we have $k = 7$.



PROCESS	1	2	3	4	5	6	7	8	9
A	H	H	H	H	H	H	H	H	H
B	H	H	L	H	H	L	H	L	H
C	H	L	L	H	L	L	L	L	H
D	L	L	L	H	H	H	H	H	H
E	L	L	L	L	L	L	H	H	H

Fig. 11. Dynamic hybrid protocol

every time the load changes in at least one process, roles are reassigned and the affected processes execute the transition algorithm on-line. It can be observed that the hybrid protocol using both types of message sources, out-performs the two other protocols, independently of individual process rates. With the Quasi-Periodic source it shows an almost constant message delivery time, with a temporary increase in message delivery latency upon each change in the operational envelope (this is due to the time required to make local decisions and the disturbance introduced by the switching protocol). With the Poisson source, although more irregular and with a slightly higher delivery time, the results are still better than either the symmetric (which performs poorly with Poisson sources[8]) or token-site protocol. In an extended report [21] simulations results obtained with more elaborate scenarios are presented.

5.5 Advantages of Topology-awareness for Total Ordering

Among the several algorithms for implementing total ordering, the *token-site* [5, 12] and *symmetric* [19, 7] are the most widely used approaches. Several token-site protocols have been optimized for local area networks [5, 22, 12, 1], so it is fair to say that LAN topology-awareness has been used with success. Previous optimizations for symmetric approaches are not based on topology [7, 18]. A hybrid approach is used by the Newtop protocol [9] where some groups can operate using

[8] Only certain values of the symmetric protocol appear in the figure so as to avoid losing detail in the token-site and hybrid plots.

a symmetric protocol and others using a token-site protocol. We have extended the latter approach by allowing both protocol types within the same group.

Our protocol is able to dynamically adapt to changes in throughput and in network delays while reducing latency through a rate-synchronization policy. The hybrid protocol was simulated for several scenarios, using different network topologies and traffic patterns. Results show that the hybrid protocol can offer significant improvements in message latency. Using simple heuristics, it is possible to make all switching decisions local to each process. Alternative heuristics, for the mode-assignment algorithm and for the switching policies, are currently being studied, to see if they can provide better performance. The protocol can be applied directly to a WAN-of-LANs network (for instance, at the site level of the NavTech model) or be used exclusively in the WAN part (at the cluster agent level). A hierarchical combination [1] with protocols specially designed for local area networks (such as xAMp [22] or Totem [18]) is possible under our model.

6 Related Work

The NavTech work in the Broadcast project 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 are focused on the solution of a particular problem. NavTech was pioneer in defining a generic architectural construct, the WAN-of-LANs model, and applying it in a vertical manner. Amongst the projects that have addressed the topology issue in a more systematic way, we will briefly mention Totem [18] and Transis [1].

The Totem system provides total order multicast over interconnected local-area networks. The system is based on a composition of *intra*-LAN and *inter*-LANs protocols. The intra-LAN protocol, or Single-Ring protocol, is a total order protocol based on the rotating-token paradigm. It is particularly well suited to Ethernet as the token mechanisms provides collision avoidance. The Inter-LAN protocol, or Multiple-Ring protocol, uses Lamport timestamps to ensure global total order. The Transis system follows a network model that has similarities with ours. The wide area network is modeled as a hierarchy of multicast clusters, and clusters are arranged in a hierarchical group structure. The work on Transis 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.

7 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.

We have designed several proof-of-concepts algorithms that are based on the WAN-of-LANs network model. We have addressed a number of different problems to assess the coverage of the model (namely, clock synchronization, causal order, total order). The applicability of the the WAN-of-LANs network model to the consensus problem is currently under study. Each of our algorithms uses the WAN-of-LANs concept in a different way, but 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.

As future work, we intend to benefit from this experience to refine our LAN-of-WANs model, clearly identifying the relevant properties that must be made visible for the benefit of the protocol designer, and which properties must be kept hidden to preserve acceptable portability.

Acknowledgments

The work on clock synchronization was done in collaboration with A. Casimiro. The work on total order protocols was done in collaboration with H. Fonseca, who also provided the simulations for the causal order protocol.

References

1. Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high-availability. In *Digest of Papers, The 22nd Int. Symp. on Fault-Tolerant Computing Systems*, pages 76–84. IEEE, 1993.
2. K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM, Transactions on Computer Systems*, 5(1), February 1987.
3. K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM, Transactions on Computer Systems*, 9(3), August 1991.
4. T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 34(1):225–267, 1996.
5. J. Chang and N. Maxemchuck. Reliable broadcast protocols. *ACM, Transactions on Computer Systems*, 2(3), August 1984.
6. B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, July 1991.
7. D. Dolev, S. Kramer, and D. Malki. Early delivery totally ordered multicast in asynchronous environments. In *Digest of Papers, The 23th Int. Symp. on Fault-Tolerant Computing*, pages 544–553. IEEE, 1993.
8. Simon Even. *Graph Algorithms*. Computer Science Press, 1979.
9. P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th Int. Conf. on Distributed Computing Systems*, pages 296–306. IEEE, 1995.
10. A. Heybey. The network simulator version 2.1. Technical report, M.I.T., September 1990.

11. P. John. *Statistical Methods in Engineering and Quality Assurance.* John Wiley & Sons Inc, 1990.
12. M. Kaashoek and A. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th Int. Conf. on Distributed Computing Systems*, pages 222–230. IEEE, 1991.
13. R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the Ninth Annual ACM Symp. of Principles of Distributed Computing*, pages 43–57, 1990.
14. L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
15. W. Lloyd and P. Kearns. Bounding sequence numbers in distributed systems: a general approach. In *Proceedings of the 10th Int. Conf. on Distributed Computing Systems*, pages 312–319, Paris, France, May 1990. IEEE.
16. S. Meldal, S. Sankar, and J. Vera. Exploiting locality in maitaining potential causality. In *Procedings of the 10th ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 231–239, 1991.
17. David Mills. Network time protocol (version 2): Specification and implementation. Technical Report RFC 1119, DARPA Network Working Group, September 1989.
18. 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 Int. Symp. on Fault-Tolerant Computing Systems*, pages 61–66. IEEE, June 1995.
19. 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.
20. M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Information processing letters*, 39(6):343–350, September 1991.
21. L. Rodrigues, H. Fonseca, and P. Veríssimo. Totally ordered multicast in large-scale systems. In *Proceedings of the 16th IEEE Int. Conf. on Distributed Computing Systems*, pages 503–510, Hong Kong, May 1996. (extended report available).
22. L. Rodrigues and P. Veríssimo. xAMp: a multi-primitive group communications service. In *Proceedings of the 11th Symp. on Reliable Distributed Systems*, pages 112–121. IEEE, 1992.
23. L. Rodrigues and P. Veríssimo. Causal separators for large-scale multicast communication. In *Proceedings of the 15th IEEE Int. Conf. on Distributed Computing Systems*, pages 83–91, Vancouver, British Columbia, Canada, May 1995. (extended report available).
24. A. Schiper, J. Eggli, and A. Sandoz. A New Algorithm to Implement Causal Ordering. In *Proceedings of the 3rd Int Workshop on Distributed Algorithms*, volume LNCS 392, pages 219–232, Nice - France, September 1989. Springer Verlag.
25. F. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):290–319, December 1990.
26. A. Schwarz and F. Mattern. Detecting Causal Relationships in Distributed Computations: In search of the Holy Grail. Technical report, Departement of Computer Science, University of Kaiserlautern, 1991.
27. M. Singhal and Kshemkalyani A. An Efficient Implementation of vector clocks. Technical report, Ohio State University, October 1990.
28. P. Stephenson. *Fast Causal Multicast.* PhD thesis, Cornell Univ., February 1991.
29. R. van Renesse, Ken Birman, and S. Maffeis. Horus: A flexible group communications system. *Communications of the ACM*, 39(4):76–83, April 1996.