Totally Ordered Multicast in Large-Scale Systems

Luís Rodrigues **Henrique Fonseca** **Paulo Veríssimo**
INESC*

Totally ordered multicast protocols have proved to be extremely useful in supporting fault-tolerant distributed applications. This paper compares the performance of the two main classes of protocols providing total order in large-scale systems: token-site and symmetric protocols. The paper shows that both classes of protocols can exhibit a latency close to $2D$, where D is the message transit delay between two processes. In face of these observations, the paper makes the following contributions: it presents a rate-synchronization scheme for symmetric protocols that exhibits a latency close to $D + t$, where t is the largest inter-message transmission time; it proposes a new hybrid protocol and shows that the hybrid scheme for heterogeneous topologies performs better than any of the previous classes of protocols in isolation; finally, the paper presents an algorithm that allows a process to dynamically adapt to changes in throughput and in network delays. The combination of these three techniques results in a dynamic hybrid scheme that, when applied to systems where the topology/traffic patterns are not known a priori, *offers a much lower latency than non-hybrid approaches.*

Keywords: Distributed Systems, Network Protocols, Multicast Communication, Total Order, Large-Scale Systems, Performance

1 Introduction

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 [23], which is a general paradigm for implement of fault-tolerant distributed applications. Although several protocols have been described in the literature [5, 3, 19, 17, 13, 12, 4, 16, 7, 1, 11], few were specifically targeted to operate in large-scale systems.

Among the several algorithms to implement total ordering, the *token-site* [5, 12] and *symmetric* [19, 7] are the most used approaches. In the token-site approach, one (or more) sites are responsible for ordering the messages on behalf of the other processes in the system. In the symmetric approach, ordering is established by all processes in a decentralized way, using information about message stability. Both methods have advantages and disadvantages.

Token-based protocols provide good performance when a single process is producing messages at a time. In this case, the producer process keeps the token and orders the messages as it sends them. However, when more than one process are transmitting, the latency is limited by the time to circulate the token or to request an order number from the token site. In large-scale

* Instituto de Engenharia de Sistemas e Computadores, R. Alves Redol, 9 - 6^o - 1000 Lisboa - Portugal, Tel.+351-1-3100000. Direct Line: 3100280, Fax: +351 1 525843.

networks, link delays are far from negligible (in fact, although throughput has been augmenting with new technologies, latency has not improved as much). Unfortunately, the message delivery latency for a process that does not hold the token is at least $2D$ (where D is the network delay), 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.

Symmetric protocols have a number of very appealing features. They are fully-decentralized and, since all processes are treated equal, they provide good load distribution. Unfortunately, symmetric protocols require that all processes send messages to enforce message stability. Additionally, although symmetric protocols provide the potential for low latency in message delivery when all processes are producing messages, we will show that such low latency is not easily attained. Perhaps surprisingly, in face of results published so far [15], when processes exhibit different message transmission rates the latency can also approximate $2D$ as in the token-site approach.

We present a new rate-synchronization scheme for symmetric protocols that can obtain a latency closer to $D + t$ (where t is the largest inter-message transmission time in the system), independently of the relation between t and D. Still, message latency is a function of the transmission rate of the slower processes in the system. Thus, symmetric protocols tend to perform poorly in environments where the majority of processes produce messages at very low rates [15].

In a large scale network, the traffic patterns of the processes are usually heterogeneous. The same applies to the properties of the links connecting the processes: some processes will be located within the same local area network, others will be connected through slow links, subject to long delays. In such an environment, none of the previous approaches can provide optimal performance.

We propose a new hybrid scheme for implementing total ordering in large-scale systems, providing message ordering between groups of processes, not necessarily between all processes in the system. In our hybrid scheme all processes multicast the messages directly to all group members. However, only certain processes are allowed to establish message order: these processes are said to operate in *active* mode. Active processes issue order numbers, also called *tickets*, for their own messages and for the messages of the processes that operate in *passive* mode. At a given instant, each passive process is associated with a single active process, which issues tickets for its messages on its behalf (however, this association may change with time). Tickets issued by different active processes are ordered using a symmetric algorithm. Thus, in our scheme, some processes order messages using a symmetric approach and others use a token-site approach in order to minimize overall message latency.

In the limit situations, the dynamic hybrid protocol will resemble either a pure symmetric or a pure token-site protocol. In an intermediate scenario, operational modes are selected according to the properties of a process and of its links. We show that for heterogeneous topologies, where links between different processes have different message transit delays, and where processes are subject to different inter-message transmission periods, a hybrid scheme performs better than any of the previous schemes in isolation.

Finally, in order to adapt to variations in the client throughput, the paper presents an algorithm that allows a process to dynamically switch between the passive and active modes. We show that the dynamic hybrid scheme can be successfully applied in systems where the topology/traffic patterns are not known *a priori*, exhibiting a lower latency than the static hybrid approach. The merits of our approach are illustrated with simulation results.

The paper is organized as follows. Section 2 surveys related work and briefly introduces the problem of providing a totally ordered multicast. Section 3 states our assumptions about the communication system and describes our simulation tool. The self-synchronizing symmetric approach is presented in Section 4. Our hybrid protocol if presented in Section 5 for static topologies. Section 6 presents the switching protocol and Section 7 shows how it is used with dynamic topologies. Concluding remarks appear in Section 9.

2 Related work

A number of algorithms to enforce total order have been published in the literature [5, 3, 19, 17, 13, 12, 4, 16, 7, 1, 11]. One solution is to rely on *logical clocks* [14], or *vector clocks* [3, 19, 13] (which can also be used to ensure causal delivery): messages are delivered according to their partial order and concurrent messages are totally ordered using some deterministic algorithm. These protocols are also known as *symmetric* protocols, since all processes execute the same algorithm. A given message can only be ordered when it is guaranteed that no more concurrent messages will arrive, meaning, in other words, when another message with larger (logical) timestamps has been received *from every sender* [23] (this implies that all senders must periodically send messages). When there is a large number of processes in the system, this scheme becomes rather cumbersome, generating heavy superfluous traffic to avoid large message delay. The ToTo [7, 17] protocol minimizes this effect through the use of a stability test that requires messages with larger timestamps only from a majority of the machines in the system, thus providing *early delivery*. However, message latency is still limited by the rate of the slowest process in the majority.

Another class of protocols is based on the selection of a special process in the system which is made responsible for establishing the orderings. This process works as a sequencer of all messages and is often called the *token* site. A number of algorithms based on this principle have been published with different degrees of fault-tolerance. The protocol family of Chang and Maxemchuk [5] rotates the token-site among the set of processes (that are both senders and recipients). The Amoeba protocol [12] is similar to that of Chang without the fault-tolerance mechanisms. ISIS [4] also uses a token-site approach but migrates the token to the process which has a higher rate. The Totem [1] protocol organizes the processors in a logical ring and circulates a token. In Totem, the token is used for several purposes: for ordering messages (messages are ordered accordingly to their "insertion" in the token) and for flow control. Other protocols, like xAMp [24, 21], use the physical order of transmission in a local-area network to establish ordering. Token protocols are appealing because they are relatively simple and provide good performance when message transit delays are small. Thus 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 processes of the system. This is a significant disadvantage for use in geographically large-scale systems.

A hybrid approach is presented in the Newtop protocol [8] which provides total order across different groups, and where some groups can operate using a symmetric protocol and others using a token-site protocol. We extend this approach by allowing both types of protocols within the same group.

Other solutions exist but will not be considered in this paper due to their performance limitation in large-scale environments. The class of algorithms known by the name of *Replica-Generated Identifiers* [3, 23] computes total order in two phases. This scheme does not scale well since it requires the collection of an acknowledgment from each recipient for every message transfer. Total ordering algorithms based on synchronized clocks (also known as Δ protocols [6]) also do not scale well because they exhibit a latency that depends on the worst case delay on the network.

3 Assumptions

In this paper we are exclusively concerned with mechanisms to provide total order delivery. Thus, for sake of clarity, we assume the availability of a reliable (unordered) multicast mechanism and its associated membership service [1]. The multicast mechanism follows the virtually-synchronous model as defined in [22] and, basically, guarantees that all messages are received by all the 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. We assume that views are *linearly* ordered ($V^i, V^{i+1}, ...$), i.e., that in case of network partitions only a majority partition remains active and continues to receive views.

In order to measure the performance of different protocols we resorted to simulation. For that purpose we used the MIT LCS Advanced Network Architecture group's network simulator, NETSIM [9]. The values presented were obtained by performing a weighted average of the maximum delivery time of each message. The simulation parameters are described below.

We assume that messages are delivered by the multicast layer with a delay that is a function of the network delay between the sender and the recipient. This network delay $D_{(s,r)}$, between a source s and a recipient r, is represented by a certain distribution function, with a mean value of $\mu_{(s,r)}$, and a variance of $\sigma^2_{(s,r)}$. Thus, if a given process s multicasts a message at real-time t, process n will receive it, on average, at real-time $t+\mu_{(s,n)}$, process m, at real-time $t+\mu_{(s,m)}$, and so on. A process receives its own messages immediately. Generally, the distribution function depends on the type of network being used to interconnect each sender-recipient pair. In this work we consider two different networks: a local area network, with small values of μ and σ^2 (in the order of 20ms and 0.05, respectively) and a wide area network, with values of μ and σ^2 in the order of 500ms and 0.3-0.5, respectively. A shifted *chi-square* distribution function was used to model both networks. Additionally, we assume that each process is subject to a different traffic load. The traffic load of each process is also described by a distribution function. In this paper, we present results obtained with a Poisson and a Quasi-Periodic message source. With the Poisson message source, the interval between each message transmission is described by a random variable with an *exponential* distribution. With the Quasi-Periodic message source, the interval between each message is described by a random variable with a very narrow *normal* distribution.

So as not to extend the length of the report, we will not discuss certain aspects related with the implementation of this protocol, such as message complexity, code complexity and ease of testing and debugging. However, based on our previous experience with the implementation of xAMp [21], we believe that the protocol presented here is not more complex than others in the bibliography.

[1] A number of recent systems [4, 18, 20] also implement total order on top of reliable multicast services.

4 Latency of symmetric protocols

We have enumerated the advantages of symmetric total ordering protocols. Being fully decentralized, they offer good failure recovery and load balancing. Additionally, these protocols have the potential for offering low latency message delivery: since a message can be delivered as soon as a message with a higher ticket is received from every other process, if all processes are producing messages at a fast rate, messages should stabilize quickly. In fact, we could hope to approximate the latency to $D + t$, where D is the network delay (the time to disseminate the message) and t is the largest inter-message transmission time of all sources (the time required for the message to become stable).

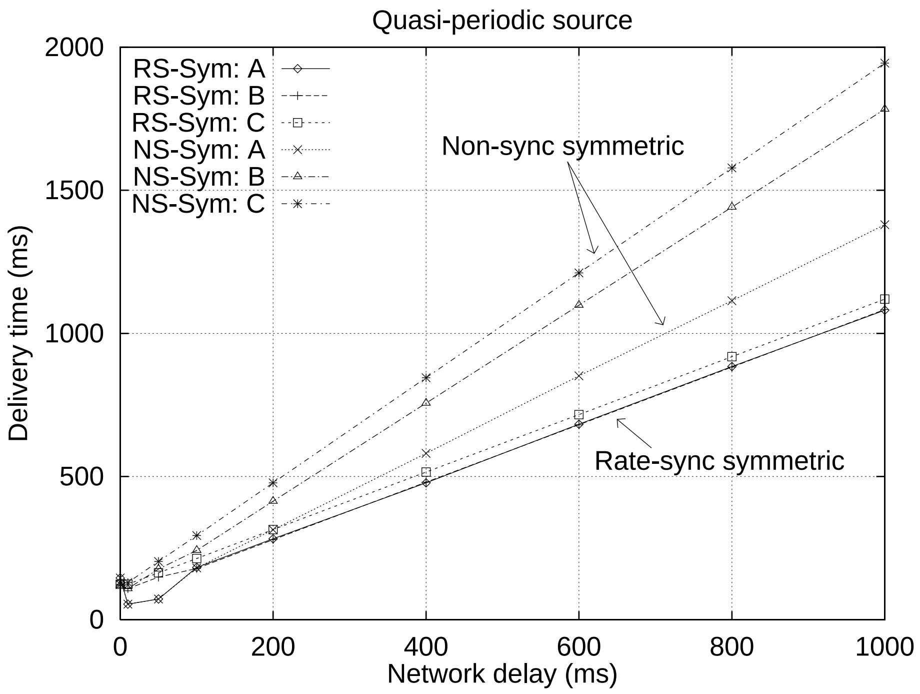
4.1 Effect of different rates

In this paper, we simulated a variant of symmetric protocols which is ssimilar to the protocols of [23, 8]. The protocol works as follows: each process has a unique identifier p_i (there is a total order on process identifiers) and keeps a Lamport logical clock [14] t_i that is updated upon send and receive events. All messages $\langle m \rangle$ are timestamped with the logical clock of the sender at transmission time. Each logical clock is updated according to the rules originally proposed by Lamport. Before a message is sent, t_i is incremented, and whenever a message $\langle m \rangle$, with timestamp t_m is received, the logical clock is set to $t_i = max(t_i, t_m)$. Finally, messages are delivered ordered by their timestamps (and messages with the same timestamp are delivered ordered by the sender's unique identifier).

When experimenting this protocol with several network delays and inter-message transmission times, we observed that, when processes had different traffic patterns, the message latency was closer to $2D$ than to $D + t$, even for values of t much smaller than D. Figure 1 shows the results of the simulations, using a combination of five processes connected to a Local Area Network, with a variable mean delay value. In this scenario, one process is subject to a high load and the other four processes are subject to a low load. The low load is 5 msg/s (200ms inter-message transmission time) and is the same in all measurements. The figure illustrates results with different values of high load, respectively: 10 msg/s (lines A), 50 msg/s (lines B) and 100 msg/s (lines C). These results were obtained for different values of network delays, depicted in the x-axis and shows the message delivery latency, depicted in the y-axis. Each line represents the maximum message delivery time for different message sources: in this case Quasi-Periodic and Poisson. The behavior is illustrated by the lines labeled as "non-sync symmetric".

This result, although somewhat counter-intuitive, can be explained. If some processes produce messages at much higher speed than others, their tickets will be soon far bigger than the tickets produced by slower processes. Slower processes only adjust their tickets when they receive a message from the faster processes (D after being sent), and the faster process will only receive a message with the updated ticked after another D delay, thus having its own messages delayed by $2D$. This behavior is illustrated in Figure 2a for a two-process scenario: process A produces messages at a faster pace and is forced to wait for the messages of the slower process B (whereas B delivers the messages as they come from A).

A possible solution for this problem would be to force slower processes to send null re-synchronization messages (this is the solution used when a process remains idle for a long time). However, this would require additional consumption of processor and network resources. We propose a new method to improve the message latency based on synchronizing the rate at which tickets are incremented, even though the processes do not produce messages at the



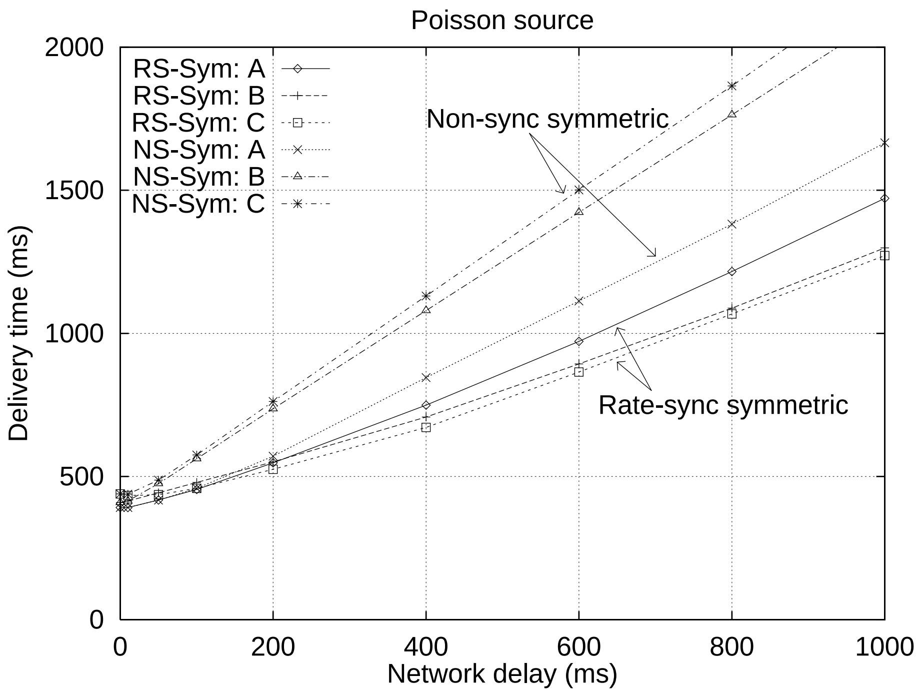


Figure 1. Non-synchronized symmetric protocol (NS-Sym) and rate-synchronized symmetric protocol (RS-Sym)

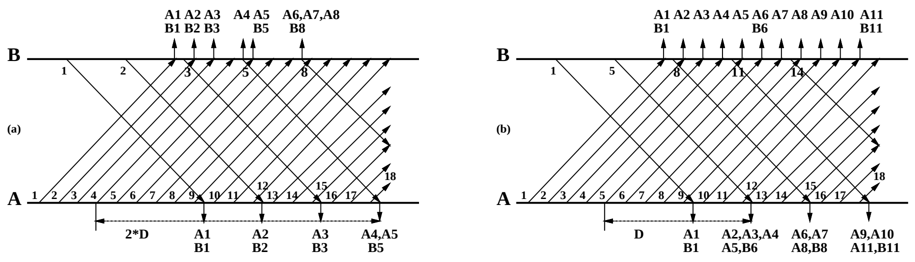


Figure 2. Rate-synchronization

same pace. This is possible because tickets have the property that, although they need to be issued with increasing values, these values do not need to be consecutive (this property is also used, for instance, in [8]). In order to decrease message latency, we make the slower processes increment the absolute value of their tickets in such a way that they increase at approximately the same rate of the tickets of the fastest process. Figure 2b illustrates the ideal behavior. Although process B is producing less messages, the tickets advance at the same pace of the tickets of process A. As it can be observed in the figure, most messages are subject to smaller delays than in the non-synchronized case; for instance, process A delivers its own messages much sooner than in the non-synchronized case. The example in the figure clearly illustrates the advantages of synchronizing the ticket generators in the case of periodic senders.

4.2 On-line synchronization

In order to make practical use of this observation, and since usually there is no *a priori* knowledge of the rate at which processes will produce messages, one has to find a technique to allow on-line synchronization of ticket generators. We experimented with a simple heuristic based on an online evaluation of the rate at which processes are transmitting and the delays in the links between the processes.

Each process timestamps every message with its own 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 the delays in the links between the processes, a simple round-trip delay method is used. At every pre-determined fixed interval of time, 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 himself and all recipients. As the symmetric protocol relies on the fact that all processes must be constantly sending messages, it is obvious that after a brief initial delay all processes will have the necessary information to compute the link delays. Special care must be taken in the choice of the time interval between null message responding: a small value gives an up-to-date and precise value of the link delay but produces a great number of null messages. As the link delay average remains more or less constant for large periods of time, our experience showed that time intervals in the order of seconds are enough for maintaining an accurate notion of the link delay.

In the simulations presented in this paper we have used a simple *mean-shift* detector: an initial mean value of rate and delay is calculated using the first seven samples from each process; whenever a run of seven or more samples fall either all above the mean value or all below it, that mean value is recalculated and used for the next iteration. This and other *mean-shift* detectors are described in detail in [10] and, as a future work, we plan to experiment with other detectors to evaluate their performance.

We denote by X_j^i the evaluation at process i of the average inter-message transmission time at process j. Also, we denote by $D_{(j,i)}^i$ the evaluation at process i of the delay between j and i. Using these values (which are updated at every message reception by the methods described above), process i can synchronize its ticket generator with the ticket generator of the fastest process as follows: every time it receives a message $\langle m \rangle$, with ticket t_m, from the process with the fastest rate f ($\forall_{q \neq f} X_f^i < X_q^i$), it sets its ticket counter, t_i, to $t_i = max(t_i, t_m + D_{(f,i)}^i / X_f^i)$.

Naturally, the result of this simple heuristic will not keep the ticket generators fully-synchronized. The results depend on the number of samples used to maintain the averages and on the distribution of traffic generators. In order to evaluate the power of the rate-synchronization technique, we have compared an implementation of a pure symmetric protocol with that of a rate-synchronized version for different traffic patterns and different relative rates. The results are also presented in Figure 1, as described before. The figure presents the latency of message delivery with and without rate-synchronization. The advantages of the rate-synchronization scheme are evident. For instance, the values obtained for the rate-synchronized protocol, using the Quasi-Periodic source, closely follow a $D + t_{\text{low-rate}}/2$ line (and they tend to $D + t_{\text{low-rate}}$ with a Poisson source).

In this section we have presented a rate-synchronization scheme that improves the message delivery latency of symmetric ordering algorithms. While the message latency of a non-synchronized protocol can be as high as $2D$, even for inter-message transmission times much smaller that D, with our rate-synchronization scheme the latency decreases to values close to $D + t_{\text{low-rate}}$ (where $t_{\text{low-rate}}$ is the inter-message transmission time of the slowest process). This means that, whenever the inter-message transmission time is smaller than D, the latency of a symmetric approach will be smaller than that of a token-site protocol (which is $2D$). On the other hand, for values of $t_{\text{low-rate}} >> D$, the token-site method still offers a lower latency. In the next section, we will show how this result can be used to create hybrid configurations, where some processes operate using the symmetric approach, while others use the token-site approach.

5 Static topologies

We present an hybrid scheme for static topologies, i.e., topologies where traffic patterns, rates and communication delays are known a priori and do not change over time. We will later expand these results to dynamic topologies. In our hybrid scheme we allow some processes to operate in the symmetric mode (these processes are said to be *active*) and other processes to operate in the 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 counter c_i for the messages it sends. Thus, each message is uniquely identified by the pair (p_i, c_i). Messages are vs-multicasted directly to all processes of the group. Active processes, keep an extra counter, the *ticket number* t_i. Ticket numbers are updated as in the symmetric protocol described in Section 4, i.e, when a message is received or sent. A ticket is a triplet $(p_i, t_i, (p_j, c_j))$ consisting of: the identification of the active process p_i; a ticket number t_i; and a message identifier (p_j, c_j). An active process issues tickets for its own messages and for the messages of its associated passive processes. At a given instant, each passive process is associated with a single active process, called the passive process *sequencer* (an active process can be a sequencer of more than one passive process). Passive processes vs-multicast their messages to all processes in the group which then wait for a ticket stating the total order of this message, ticket that will be sent by the passive process's *sequencer*. In order to be disseminated to all processes, tickets are piggy-backed in the messages sent by the active processes. Tickets are ordered as in the 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 the network delays between processes. In order to configure the system, we use a heuristic that analyses each pair of processes in isolation. Consider a process n, subject to a load characterized by an inter-message transmission time of 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, the inter-message transmission time is longer than the round-trip delay of the active process 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 n 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 less latency but provides better load distribution).

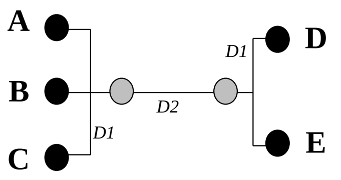
```
forall process $n$ set $n$ mode to passive
let $a$ be the process with highest rate; set $a$ mode to active; set changed to true
while (changed is true) do // iteration
    set changed to false
    forall $j$ such that $j$ mode is passive do
        let $a$ be the active process closest to $j$
 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

Figure 3. Mode assignment heuristic

The complete algorithm to assign roles can be obtained by applying the previous rule recursively, as described in Figure 3. Initially, all processes are considered 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 re-executed until there is an interaction where no change is made to the network.

In order to test the effectiveness of our approach, we compared under different scenarios the performance of the hybrid protocol with a pure symmetric algorithm and a token-site algorithm. The pure symmetric algorithm we have used is the one described in Section 4. The pure token-site algorithm used for comparison terms was the non fault-tolerant version of [12], where a single site issues tickets on behalf of all other processes in the group.

The results are illustrated in Figure 4. The results were obtained for a system



PROCESS									
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*
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

SCENARIO

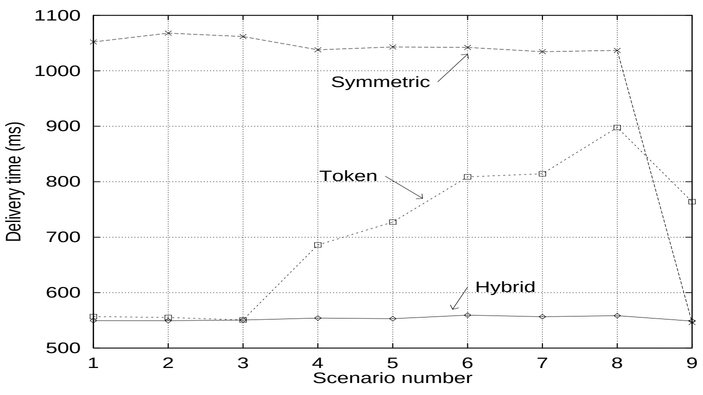


Figure 4. Static hybrid protocol

of five processes, connected by a network as shown also in the figure (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). We have simulated nine different scenarios, which differ in the 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 a high load, process D has a low load, and so on). In this case we have used $D1 = 20ms$, $D2 = 500ms$, a high rate of $100msg/s$ and a low rate of $1msg/s$ (both with a Quasi-Periodic source). Figure 4 depicts the performance of the three protocols for each scenario (in the token-site protocol, process A holds the token in all scenarios). For the hybrid case, heavily loaded processes are active and lightly loaded processes are passive in all scenarios. The token-site protocol offers the lowest latency when all high-load processes are close to the sequencer and clearly degrades when the load shifts to distant processes. The symmetric protocol offers the lowest latency in scenario 9, where all processes have a high load.

The almost flat line represents the performance of the hybrid protocol. Since in the limit scenarios, the hybrid approach resembles a pure token or a pure symmetric protocol, the performance of the hybrid protocol matches these best cases. Additionally, in all the intermediate cases where pure symmetric or token-site protocols degrade their performance, the hybrid protocol continues to offer extremely good performance.

6 Mode switching protocol

In order to apply the hybrid scheme to dynamic topologies, we need an algorithm that allows a process to dynamically switch between active and passive mode. This section describes such 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 some active processes, which are acting as sequencers of other processes, crash. In this case, the 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 a correct operation, all correct processes in the system must see the same sequence of configurations. Thus, the order in which transitions are executed, with regard to the message flow and membership indications, must be agreed before transitions actually take place. In order to reach agreement about the $(i+1)$th configuration we use the properties of the underlying view-synchronous layer (vs-layer) and the total order of messages established by the ith configuration. For self containment, we include the vs-multicast definition [22]:

vs-**multicast:** *Consider a set of processes g, a view $V^i(g)$, and a message m multicast to the members of group $V^i(g)$. The multicast m is* vs-*multicast in view $V^i(g)$ iff: if $\exists_p \in V^i(g)$ which has delivered m in view $V^i(g)$ and has installed view $V^{i+1}(g)$, then all processes $q \in V^i(g)$ which have installed $V^{i+1}(g)$ have delivered m before installing $V^{i+1}(g)$.*

The advantage of the vs-layer is the guarantee that, in case of failures, all surviving processes receive the same set of messages before the 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 the protocols presented next, that will implement the hybrid scheme. However, in the switching protocol we make no assumptions about the consistency of rates and network delay evaluations. Thus, no process can assume that some other process will change state just because such a transition is plausible according to its own local information. That is, 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, we first present some definitions. 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 the process changes roles). We define a *system configuration*, $C = \{V^i, \bigcup_{j \in V^i} D_j\}$, as a system view plus the process descriptors of all processes in the view. We also assume that each process j, keeps a record of the last of its own messages that has been delivered, l_j. Finally, we assume that a passive process p keeps the process descriptor of its sequencer in a variable called $S(p)$.

A pseudo-code description of the protocol is presented in Figure 5. The following text presents an informal description of the protocol functionality.

6.1 Initial configuration

When the hybrid protocol starts all processes must agree on some initial configuration. The exact configuration is not relevant since the system is able to reconfigure itself (as long as there exists at least one active process in the initial configuration). In all our simulations, we used an initial configuration where all processes are active and remain in such state until they have received enough messages to evaluate the traffic load and network delays.

6.2 Operation in steady-state

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

Since messages can be transmitted concurrently with events that generate configuration changes, it is possible that the assigned sequencer fails or increments 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: $\langle \textit{reassign}, p_i,]l_i, c_i], D_{\text{new}}\rangle$, where p_i is the identification of the source, $]l_i, c_i]$ a range of message sequence numbers and D_{new} the new sequencer for those messages. As it will be seen, reassign messages are only sent when the selected sequencer fails or becomes passive.

If a passive process fails, all of his messages delivered by the vs-layer before the view change but not yet ordered by his sequencer, are silently discarded by all processes. This procedure is safe, because the properties of the vs-layer guarantees that the tickets for those messages are also 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 in the pending queue that are assigned to them (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. Ticket numbers are updated according to the rate-synchronization technique described in Section 4. Although only active processes issue tickets, all processes (including passive processes) keep their ticket numbers synchronized: 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 $\langle \textit{reassign}, p_i,]l_i, c_i], D_s\rangle$ is never delivered: it is just used to update the sequencer field of all specified messages in the pending queue.

6.3 Events that trigger reconfigurations

Process mode transitions and process crashes induce a sequence of system configurations. In order to change their modes, processes broadcast special messages, namely $\langle goingToActive, p_a, c_a, D_s\rangle$ and $\langle goingToPassive, p_a, c_a, D_s\rangle$ messages. Such messages are sent in total order as any other data message. The delivery of such messages triggers the installation of a new system configuration. More precisely, there are three types of events that may change the system configuration:

```
code for process $i$

Declare Types
  Role    oneof (active, passive);
  State   oneof (active, chgseq, topassive, passive, toactive);
Declare Variables
  ProcessIdentifier   $p_i$;
  Integer             $c_i$;      // the message count
  Integer             $l_i$;      // last delivered
  Role                $r_i$;
  Integer             $rn_i$;     // role number
  State               $state_i$;
  Integer             $t_i$;      // ticket counter
  ProcessDescriptor   $S_i$;      // my sequencer
  Configuration       $C_i$;      // current configuration
  MessageQueue        $Q_i$;      // pending messages
  ListOfTickets       $UT_i$;     // list of unordered tickets
  ListOfTickets       $OT_i$;     // list of ordered tickets
  ListOfTickets       $LT_i$;     // list of issued tickets
  // $T(M)$ denotes the ticket issued for message $M$

declare function issueTickets
  forall message $M = \langle mtype, p_j, c_j, D_M, bits \rangle \in Q_i$ do
    if ($D_M = (p_i, r_i, rn_i) \wedge T(M) \notin LT_i$) then
      issue a ticket $T(M)$ for M (update $t_i$);
      $LT_i := LT_i \cup \{T(M)\}$;
    fi;
  od; // forall

declare function deliverMessage (Message M)
  if $M = \langle data, p_j, c_j, S, Muser \rangle$) then // data message
    deliver Muser;
    if $(p_j = p_i)$ then
      $l_i := c_j$;
      if ($state_i = chgseq \wedge l_i = c_i$) then
        $S_i :=$ selectNewSequencer();
        $state :=$ passive; // re-start sending
      fi;
    fi;
  fi
  if $M = \langle goingToActive, p_j, c_j, S \rangle$ then
    $C_i :=$ new configuration;
    if $(p_j = p_i)$ then
      $rn_i := rn_i + 1; l_i := c_j$;
      $state_i :=$ active; $S_i := (p_i, r_i, rn_i)$;
      call issueTickets; // re-start sending
    fi;
  fi;
  if $M = \langle goingToPassive, p_j, c_j, S \rangle$ then
    $C^{temp} :=$ new configuration;
    if (no active process in $C^{temp}$) then // abort
      if $(p_j = p_i)$ then
        $l_i := c_j$; $state_i :=$ active; $S_i := (p_i, r_i, rn_i)$;
        call issueTickets; // re-start sending
      fi;
    else
      $C_i := C^{temp}$;
      if $(p_j = p_i)$ then
        $rn_i := rn_i + 1; l_i := c_j$;
        $state_i :=$ passive; // re-start sending
      fi;
      if ($state_i \neq$ active) then
        $l_i := c_j$;
        set $S_i$ to Descriptor of closest active process;
        vs-multicast($[]\langle reassign, p_i, ]l_i, c_i], S_i \rangle$);
      fi;
    fi;
  fi;

declare function deliverInOrder
  while ( $T(M) =$ queueHead ($OT_i$) $\wedge M \in Q_i$ ) do
    remove $T(M)$ from $OT_i$;
    remove $M$ from $Q_i$;
    call deliverMessage (M);
  od;

main loop:  wait event
  when $[tickets]\langle M \rangle$ received from process $j$ do
    if ($[tickets] \neq \emptyset$) then
      $UT_i := UT_i \cup [tickets]$;
      move tickets in order from $UT_i$ to $OT_i$;
    fi;
    update $t_i$ using rate-synchronization;
    if $M = \langle reassign, p_j, ]f, t], D_{new} \rangle$ then
      forall $c \in ]f, t]$ do
        $Mc := \langle anytype, p_j, c, D_{Mc}, bits \rangle \in Q_i$;
        change $D_{Mc}$ to $D_{new}$ in $Mc$;
      od;
    else
      $Q_i := Q_i \cup \{\langle M \rangle\}$;
    fi;
    call deliverInOrder;
    if ( $state_i =$ active) then call issueTickets; fi;
  od;

  when there is a message Muser to send and $state_i =$ passive do
    $c_i =: c_i + 1$;
    vs-multicast ($[]\langle data, p_i, c_i, S_i, Muser \rangle$);
  od;

  when there is a message Muser to send and $state_i =$ active do
    $c_i =: c_i + 1$;
    $M := \langle data, p_i, c_i, D_i, Muser \rangle$;
    issue ticket $T(M)$ for $M$ (update $t_i$);
    $LT_i := LT_i \cup \{T(M)\}$;
    vs-multicast ($[LT_i]\langle M \rangle$);
    $LT_i := \emptyset$;
  od;

  when $state_i =$ active and time to go to passive do
    $c_i := c_i + 1$;
    $M := \langle goingToPassive, p_i, c_i, S_i \rangle$;
    issue ticket $T(M)$ for $M$ (update $t_i$);
    $LT_i := LT_i \cup \{T(M)\}$;
    $state_i :=$ topassive; // stop sending
    vs-multicast ($[LT_i]\langle M \rangle$);
    $LT_i := \emptyset$;
  od;

  when $state_i =$ passive and time to change sequencer do
    $state_i :=$ chgseq; // stop sending
  od;

  when $state_i =$ passive and time to go to active do
    $c_i := c_i + 1$;
    $M := \langle goingToActive, p_i, c_i, S_i \rangle$;
    $state_i :=$ toactive; // stop sending
    vs-multicast ($[]\langle M \rangle$);
  od;

  when view $V^i$ is delivered do
    $C^{temp} := V^i, \bigcup_{j \in V^i} D_j \in C_i$;
    if ($S_i \notin C^{temp} \wedge \exists_{j \in C^{temp}} : r_j =$ active) then
      $S_i :=$ process descriptor of closest active process;
      vs-multicast($[]\langle reassign, p_i, ]l_i, c_i], S_i \rangle$);
    fi;
    if ( $\nexists_{j \in C^{temp}} : r_j =$ active) then
      select active process $m$;
      $r_m :=$ active;
      $rn_m := rn_m + 1$; // update $C^{temp}$
      if $(m = i)$ then
        // I'm my own sequencer now
        $state_i :=$ active; $S_i := (p_i, r_i, rn_i)$;
        vs-multicast($[]\langle reassign, p_i, ]l_i, c_i], S_i \rangle$)
        call issueTickets;
      fi;
      $C_i := C^{temp}$;
    fi;
forever; // main loop
```

Figure 5. Pseudo-code description of the hybrid protocol.

View changes. Assume that the system is in configuration $C^n = \{V^i, \bigcup_{j \in V^i} D_j\}$ and V^{i+1} is delivered by the vs-layer. A new configuration $C^{n+1} = \{V^{i+1}, \bigcup_{j \in V^{i+1}} D_j\}$ is created.
If there is no process active in such configuration (i.e, all active processes have failed) the process, m, with 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 the following reassign message:$\langle$*reassign*$, p_p,]l_p, c_p], S(p)\rangle$.

Delivery of a $\langle goingToActive, p_a, c_a, D_s\rangle$ message. Such message is sent by a passive process when it wants to become active and is delivered in total order as any other data message. Assume that the system is in configuration C^n when this message is delivered. A new configuration C^{n+1} is created by setting r_a = *active*, and incrementing rn_a. Then, C^{n+1} is installed.

Delivery of a $\langle goingToPassive, p_a, c_a, D_s\rangle$ message. Such message is sent by a active process when it wants to become passive and is delivered in total order as any other data message. Assume that the system is in configuration C^n when this message is delivered. If p_a is the unique active process in the configuration, the message is discarded and no new configuration is installed, because at least one active process must always be present in any configuration. Otherwise a new configuration C^{n+1} is created by setting r_a = *passive*, and incrementing rn_a. Then, C^{n+1} is installed.
As in the View Changes case, if there is a passive process such that $S(p) \notin C^{n+1}$, this process selects a new sequencer m and sets $S(p) = (p_m, r_m, rn_m)$. Also, it sends a reassign message:$\langle$*reassign*$, p_p,]l_p, c_p], S(p)\rangle$.

6.4 Change of sequencer

A passive process can switch its sequencer if some other process becomes active and that process is closer than the previous sequencer. Since the data messages specify the desired sequencer, switching of sequencer is very simple. To avoid disturbances in the FIFO ordering, the 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 re-starts transmitting messages.

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

6.5 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 $\langle$*goingToPassive*$, p_i, c_i, D_s\rangle$ and stops transmitting (and stops issuing tickets, even for messages assigned to it). It then waits for its own message to be delivered. When it delivers its own $\langle$*goingToPassive*$\rangle$ message it checks if it is the last active process in the group. Note that since several processes can decide to become passive concurrently, all actives might try to become passive but, since at least one active process must exist, the last one will fail. In the case it is the last active process, it simply aborts the transition and re-starts sending messages and issuing tickets. Otherwise, the new configuration is installed, and it re-starts sending messages in the passive mode.

6.6 Transition from passive to active

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

In the first case, the passive process i broadcasts a special $\langle \mathit{goingToActive}, p_i, c_i, D_s \rangle$ and stops sending messages. Then, it waits until this special message is ordered by his sequencer and delivered. When it is delivered, a new configuration is installed, as described in Subsection 6.3. All the messages sent after this new configuration are ordered by process i itself (we recall that although only active processes issue tickets, passive processes keep their ticket numbers rate-synchronized).

In case of failure of the unique active process, the passive process becomes active as soon as it receives the failure indication from the virtually synchronous layer. Upon this transition, the new active process i issues tickets for all the messages it has sent but that were not ordered in previous configurations, i.e., for all messages with sequence numbers in the interval $]l_i, c_i]$.

7 Dynamic topologies

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 be temporarily unavailable). In the previous section, we described the algorithms that allow a process to change its operational mode. With the dynamic hybrid scheme the protocol is able to automatically adapt the operational mode of each process to the changes in the operational envelope, i.e, to changes both in traffic patterns and in communication links.

When introducing the on-line rate-synchronization (Subsection 4.2), we described the techniques that allow a process to evaluate system parameters as traffic load and network delays. In Section 5 we showed how an external observer can apply this knowledge to assign roles to each process in an hybrid configuration. Since such an external observer can only be approximated, and to avoid centralized solutions, we now present a heuristic that allows each process to make a local switching decision based on its own evaluation of the system state.

The heuristic is as follows: each process keeps track of his own message rate and of the network delay between him and all other active processes. If his inter-message transmission rate is smaller than the delay to the closest active, he should himself switch to active mode, as we have already shown. If, on the other hand, his inter-message transmission rate is higher than the delay to the closest active, he should switch to passive mode using that closest active as his sequencer process. 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 taken when the difference between these values becomes greater than a given threshold (we used a threshold of 20% of the delay value).

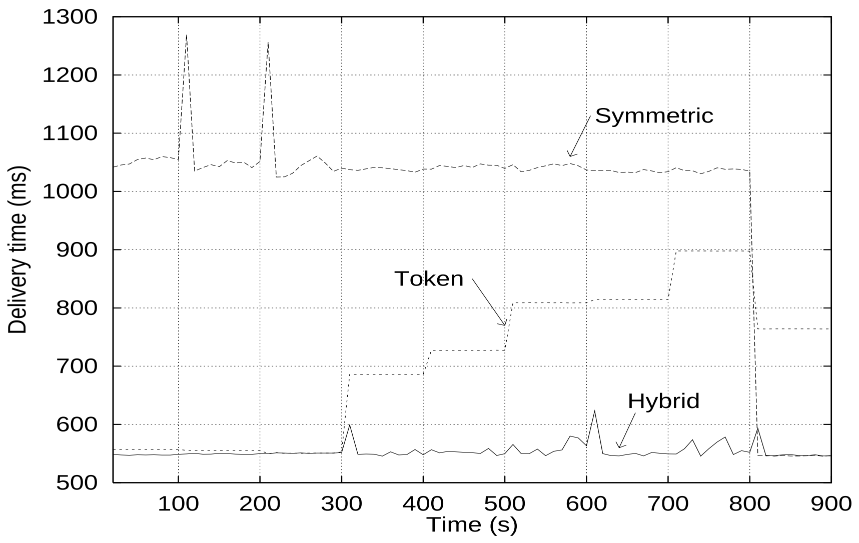


Figure 6. Dynamic hybrid protocol

The results obtained using the heuristic above are presented in Figure 6, alongside with the results from pure symmetric and token-site protocols. For clarity of exposition, we used exactly the same scenarios as in Figure 4, except that now the system ran continuously while the load of processes changed in time, making the system evolve through all nine scenarios in sequence. In the hybrid approach, every time the load changes in at least one process, the roles are reassigned and the affected processes execute the transition algorithm on-line. It can be seen that, upon each change in the operational envelope, there is a temporary increase in message delivery latency. This is due to the time required to make a local decisions plus the disturbance introduced by the switching protocol. Nevertheless, we observe that overall the hybrid protocol out-performs the two other protocols, showing an almost constant message delivery time, independently of individual process rates.

Throughout this paper, we have illustrated the concepts of the hybrid protocol with some relatively simple examples. In the next section we will present simulations results obtained with more elaborate scenarios.

8 Complex scenarios

Communication in large-scale systems is likely to involve several different speed links, with routers and gateways that will eventually slow the exchange of messages between nodes.

To evaluate the behaviour of our hybrid protocol, we designed such a scenario consisting of over a dozen nodes spead out through a few low-speed links recreating a nation-wide cluster of nodes linked to a distant inter-continental remote cluster.

In Table 1 we present the results over this complex scenario using a static topology, meaning that all nodes transmit at a constant rate and do not change this throughout the simulation. In the hybrid protocol the assignment of roles to each process was chosen statically using the algorithm presented before, like

	Hybrid	Token-site	Symmetric
Poisson	647	1034	1839
Quasi-Periodic	727	1034	1096

Table 1. Average delivery times in ms (no node tx-rate change)

it was done in Section 5. We also observed that for the token-site protocol the values measured very much depend on the location of the token sequencer.

	Hybrid	Token-site	Symmetric
Quasi-Periodic	753	1010	2211

Table 2. Average delivery times in ms (with node tx-rate change)

In Table 2 we use the same scenario but now with a dynamic topology where transition rates change constantly. In the hybrid protocol process roles follow the local heuristic presented in the previous section.

We believe that the scenarios and results presented are representative: the hybrid protocol offers significant advantages over pure symmetric or token-site approaches, in large-scale systems and heterogeneous environments. Nevertheless, we intend to perform further simulations using different types of network configurations. Even if some cases don't present significant advantages over other approaches (such as the limit cases presented before), the number of situations where performance is enhanced clearly justifies the protocol and the work developed.

9 Conclusions and future work

This paper proposed a new hybrid scheme for implementing totally ordered multicasts in geographically large-scale systems, using a combination of symmetric and token-site based protocols. This protocol is able to dynamically adapt to changes in throughput and in network delays while reducing latency through a rate-synchronization policy. Although it requires a few additional messages for evaluating link delays, the rate-synchronization policy offers by its own significant latency gains in various configurations.

We have simulated the hybrid protocol for a large number of scenarios, using different network topologies and traffic patterns. The results show that the hybrid protocol can offer significant improvements in message latency. Using simple heuristics, it is possible to make all switching decision local to each process. We are currently studying alternative heuristics, for the mode-assignment algorithm, for the rate-synchronization and for the switching policies, to see if they can provide better performance.

Although we have illustrated the benefits of the approach with a given protocol, the underlying principles of our approach could be applied to other protocols. We will experiment the rate-synchronization optimization with the early-delivery protocol described in [7].

It has been shown in [2] that total order protocols can be combined in a hierarchical manner. An optimized solution for topologies based on interconnected LANs could use a hierarchical combination of protocols specially designed for local area networks (as AMp [24]/xAMp [21] or Totem [1]) with the dynamic hybrid protocol presented here.

Acknowledgments

The authors are grateful to P. Antunes, A. Casimiro, F. Cosquer, S. Melro and M. Sequeira for their comments on a previous version of this paper.

References

[1] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 551–560, Pittsburgh, Pennsylvania, USA, May 1993.

[2] 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, 1992.

[3] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM, Transactions on Computer Systems*, 5(1), February 1987.

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

[5] J. Chang and N. Maxemchuck. Reliable broadcast protocols. *ACM, Transactions on Computer Systems*, 2(3), August 1984.

[6] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Digest of Papers, The 15th International Symposium on Fault-Tolerant Computing*, Ann Arbor-USA, June 1985. IEEE.

[7] D. Dolev, S. Kramer, and D. Malki. Early delivery totally ordered multicast in asynchronous environments. In *Digest of Papers, The 23th International Symposium on Fault-Tolerant Computing*, pages 544–553, Toulouse, France, June 1993. IEEE.

[8] 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*, Vancouver, British Columbia, canada, May 1995. IEEE.

[9] A. Heybey. The network simulator version 2.1. Technical report, M.I.T., September 1990.

[10] P. W. John. *Statistical Methods in Engineering and Quality Assurance*. John Wiley & Sons Inc, 1990.

[11] T. Johnson and L. Maugis. Two approaches for high concurrency in multicast-based object replication. Technical Report 94-041, Department of Computer and Information Sciences, University of Florida, 1994.

[12] M. Kaashoek and A. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 222–230, Arlington, Texas, USA, May 1991. IEEE.

[13] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Lazy replication: Exploiting the semantics of distributed services. Technical Report MIT/LCS/TR-84, MIT Laboratory for Computer Science, 1990.

[14] L. Lamport. Time, clocks and the ordering of events in a distributed system. *CACM*, 7(21), July 1978.

[15] L. Mahlis, W. Sanders, and R. Schlichting. Analytic performability evaluation of a group-oriented multicast protocol. Technical report, University of Arizona, 1992.

[16] K. Marzullo, S. Armstrong, and A. Freier. Multicast transport protocol. Technical report, 1992. Internet RPC 1301.

[17] P. Melliar-Smith, L. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and distributed systems*, 1(1):17–25, January 1990.

[18] S. Mishra, L. Peterson, and R. Schlichting. Protocol modularity in systems for managing replicated data. In *Proceedings of the Second Workshop on the Management of Replicated Data*, pages 78–81, Monterey, California, November 1992. IEEE.

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

[20] R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. The Horus system. Technical report, Cornell University, July 1993.

[21] L. Rodrigues and P. Veríssimo. xAMp: a multi-primitive group communications service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 112–121, Houston, Texas, October 1992. IEEE. INESC AR/66-92.

[22] A. Schiper and A. Ricciardi. Virtually-synchronous communication based on a weak failure suspector. In *Digest of Papers, The 23th International Symposium on Fault-Tolerant Computing*, pages 534–543, Toulouse, France, June 1993. IEEE.

[23] F. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):290–319, December 1990.

[24] P. Veríssimo, L. Rodrigues, and M. Baptista. AMp: A highly parallel atomic multicast protocol. In *Proceedings of the SIGCOM'89 Symposium*, Austin-USA, September 1989. ACM.