A dynamic hybrid protocol for total order in large-scale systems*

Luís E.T. Rodrigues
Instituto Superior Técnico
INESC
ler@inesc.pt

Henrique Fonseca
Universidade de Lisboa
INESC
hjf@inesc.pt

Paulo Veríssimo
Universidade de Lisboa
INESC
paulov@inesc.pt

Abstract

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) and proposes a new dynamic hybrid protocol that, when applied to systems where the topology / traffic patterns are not known *a priori*, offers a much lower latency than any of the previous classes of protocols in isolation.

Keywords: *Network Protocols, Multicast Communication, Total Order, Large-Scale Systems.*

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 [22],

*This work was partially supported by the CEC, through Esprit Project BR 6360 (Broadcast). Selected sections of this report were published in the proceedings of the 16th International Conference on Distributed Computing Systems, Hong Kong, May, 1996.

which is a general paradigm for implementing fault-tolerant distributed applications. Although several protocols have been described in the literature [2, 3, 4, 5, 7, 11, 12, 13, 15, 16, 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. A new dynamic hybrid protocol for implementing total ordering in large-scale systems is proposed. In this protocol, some processes order messages using a symmetric approach and others use a token-site approach in order to minimize overall message latency. So as to adapt to variations in client throughput or in network delay, the protocol allows processes to dynamically switch between passive and active modes. This paper shows that the dynamic hybrid protocol can be successfully applied to systems where topology / traffic patterns are not known *a priori*. Simulation results illustrate the merits of dynamic protocol.

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 the assumptions about the communication system and describes the simulation tool. The hybrid protocol is presented in Section 4 for static topologies. Section 5 presents the switching protocol and Section 6 shows how it is used with dynamic topologies. Concluding remarks appear in Section 7.

2 Related work

Among the several algorithms for implementing total ordering, the *token-site* [5, 12] and *symmetric* [18, 7] are the most used approaches[1].Both methods have ad-

[1]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 as *Replica-Generated Identifiers* [3, 22] 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 the passage of time (also known as Δ protocols [6]) also do not scale well because they may exhibit a latency that depends on the worst case network delay.

vantages 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. A number of algorithms based on this principle have been published providing 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 without the fault-tolerance mechanisms. Isis [4] also uses a token-site approach but migrates the token to the process which has a higher transmission rate. The Totem [2] protocol organizes processes in a logical ring and circulates a token that is used for ordering messages (messages are ordered accordingly to their "insertion" in the token) and for flow control. Other protocols, like xAMp [24, 20], 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 (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* [3, 18, 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. In fact, using a technique called rate-synchronization [19], 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* processes must send messages at a high rate to achieve low protocol latency. The ToTo [7, 16] 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.

A hybrid approach is used by 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. This approach is extended in this paper, by allowing both protocol types within the same group.

3 Assumptions

This paper is exclusively concerned with mechanisms for providing total order delivery. Thus, for the sake of clarity, availability of a reliable (unordered) multicast mechanism and its associated membership service is assumed[2]. We further assume that the multicast mechanism follows the virtually-synchronous model as defined in [21] 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 partitions only a majority partition remains active and continues to receive views.

In order to measure protocol performance we resorted to simulation. For that purpose, MIT LCS Advanced Network Architecture group's network simulator (NETSIM [9]) was used. Values presented here were obtained by performing a weighted average of each message's maximum delivery time. Simulation parameters are described below.

Messages are delivered by the multicast layer with a delay that is a function of network delay $D_{(s,r)}$ between the sender s and the recipient r. Network delay is represented by a probability 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,

[2] A number of recent systems [4, 17, 23] also implement total order on top of reliable multicast services.

process m will receive it, on average, at real-time $t + \mu_{(s,m)}$, process n, at real-time $t+\mu_{(s,n)}$, and so on. A process receives its own messages immediately. Usually, the distribution function depends on the type of network being used to interconnect each sender-recipient pair. Two different networks are considered: a local area network, with small values of μ and σ^2 (20ms and 0.05, respectively) and a wide area network, with large values of μ and σ^2 (500ms and 0.3-0.5, respectively). A shifted χ^2 distribution function was used to model both networks. Additionally, it is assumed that each process is subject to a different traffic load. The traffic load of each process is also described by a distribution function. Present results were obtained using a Quasi-Periodic and a Poisson message source[3].

4 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.

4.1 Protocol operation

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 multicasted, 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 referred to in Section 4.2. A ticket is a

[3]Using a Quasi-Periodic message source, the interval between each message is described by a random variable with a very narrow normal distribution. Using Poisson message source, the interval between each message transmission is described by a random variable with an exponential distribution.

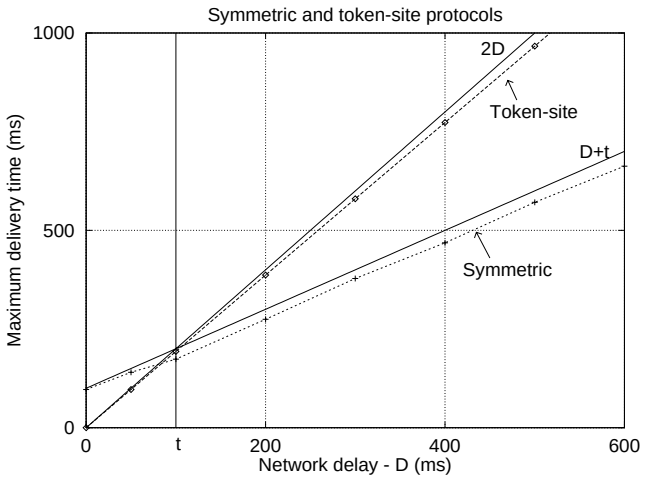


Figure 1: Latency of symmetric and token-site protocols in isolation.

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 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.

4.2 Criteria for mode assignment

In order to illustrate the behavior of both protocols in isolation we have selected the rate-synchronized symmetric protocol of [19] 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 1, 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

```
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
```

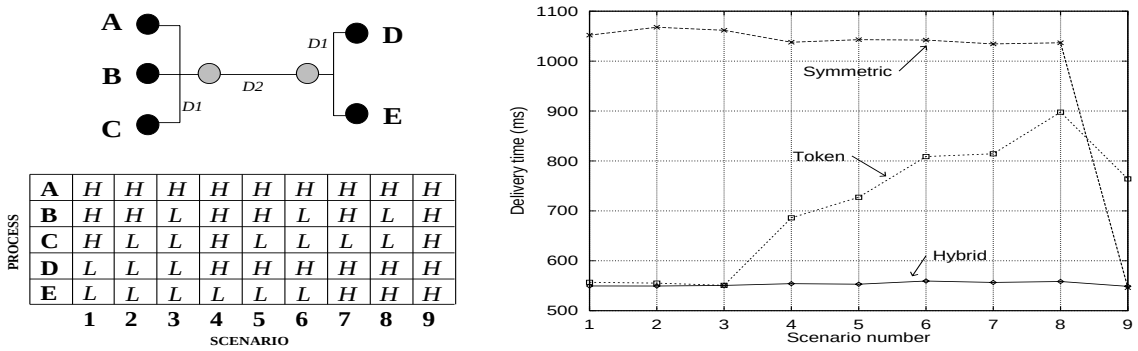
Figure 2: Mode assignment heuristic

are more favorable when $2D < D + t \equiv D < t$, and that symmetric protocols are more favorable otherwise. The next section shows how this observation can be used to assign operational modes in the hybrid protocol.

4.3 Mode assignment heuristic

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 analyses 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 2. Initially, all processes are made passive. Since at least one active process must exist in the system, the process



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

Figure 3: Static hybrid protocol

(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.

4.4 Performance with static topologies

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 4.2. The results are shown in Figure 3. They were obtained with a system of five processes, connected by a network as shown (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). Figure 3 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, when all processes have high load.

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

5 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. For self containment, the *vs*-multicast definition [21] is included:

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 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 $S(p)$.

A protocol pseudo-code description opresented in Figure 4. The following text presents an informal description of the protocol's functionality.

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

5.2 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: $\langle \textit{type}, p_i, c_i, S_i, \textit{user-data} \rangle$, 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. As will be seen, 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 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 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 [19]: 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, message $\langle \textit{reassign}, p_i,]l_i, c_i], S_i \rangle$ is never delivered: it is only used to update the sequencer field of all specified messages in the pending queue.

5.3 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 $\langle goingToActive, p_i, c_i, S_i \rangle$ and $\langle goingToPassive, p_i, c_i, S_i \rangle$ messages. Such messages are sent in total order as any other data message, 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.

5.3.1 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 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 $\langle \textit{reassign}, p_p,]l_p, c_p], S(p) \rangle$.

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 = ⟨mtype, p_j, c_j, D_M, bits⟩ ∈ Q_i do
    if (D_M = (p_i, r_i, rn_i) ∧ T(M) ∉ LT_i) then
      issue a ticket T(M) for M (update t_i);
      LT_i := LT_i ∪ {T(M)};
    fi;
  od; // forall

declare function deliverMessage (Message M)
  if M = ⟨data, p_j, c_j, S, Muser⟩) then // data message
    deliver Muser;
    if (p_j = p_i) then
      l_i := c_j;
      if (state_i = chgseq ∧ l_i = c_i) then
        S_i := selectNewSequencer();
        state := passive; // re-start sending
      fi;
    fi;
  fi
  if M = ⟨goingToActive, p_j, c_j, S⟩ 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 = ⟨goingToPassive, p_j, c_j, S⟩ 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 ≠ active) then
        l_i := c_j;
        set S_i to Descriptor of closest active process;
        vs-multicast([]⟨reassign, p_i, ]l_i, c_i], S_i⟩);
      fi;
    fi;
  fi;

declare function deliverInOrder
  while ( T(M) = queueHead (OT_i) ∧ M ∈ Q_i ) do
    remove T(M) from OT_i;
    remove M from Q_i;
    call deliverMessage (M);
  od;

main loop:  wait event
  when [tickets]⟨M⟩ received from process j do
    if ([tickets] ≠ ∅) then
      UT_i := UT_i ∪ [tickets];
      move tickets in order from UT_i to OT_i;
    fi;
    update t_i using rate-synchronization;
    if M = ⟨reassign, p_j, ]f, t], D_new⟩ then
      forall c ∈]f, t] do
        Mc := ⟨anytype, p_j, c, D_Mc, bits⟩ ∈ Q_i;
        change D_Mc to D_new in Mc;
      od;
    else
      Q_i := Q_i ∪ {⟨M⟩};
    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 ([]⟨data, p_i, c_i, S_i, Muser⟩);
  od;

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

  when state_i = active and time to go to passive do
    c_i := c_i + 1;
    M := ⟨goingToPassive, p_i, c_i, S_i⟩;
    issue ticket T(M) for M (update t_i);
    LT_i := LT_i ∪ {T(M)};
    state_i := topassive; // stop sending
    vs-multicast ([LT_i]⟨M⟩);
    LT_i := ∅;
  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 := ⟨goingToActive, p_i, c_i, S_i⟩;
    state_i := toactive; // stop sending
    vs-multicast ([]⟨M⟩);
  od;

  when view V^i is delivered do
    C^temp := V^i; ⋃_{j∈V^i} D_j ∈ C_i;
    if (S_i ∉ C^temp ∧ ∃_{j∈C^temp} : r_j = active) then
      S_i := process descriptor of closest active process;
      vs-multicast([]⟨reassign, p_i, ]l_i, c_i], S_i⟩);
    fi;
    if (∄_{j∈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([]⟨reassign, p_i, ]l_i, c_i], S_i⟩)
        call issueTickets;
      fi;
      C_i := C^temp;
    fi;
  forever; // main loop
```

Figure 4: Pseudo-code description of the hybrid protocol.

5.3.2 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 $\langle \textit{goingToPassive}, p_i, c_i, S_i \rangle$, 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 $\langle \textit{goingToPassive} \rangle$ 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.

5.3.3 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 $\langle \textit{goingToActive}, p_i, c_i, S_i \rangle$ 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 $]l_i, c_i]$.

5.3.4 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 message $\langle \textit{reassign}, p_p,]l_p, c_p], S(p) \rangle$ instructing the new sequencer to order unordered messages.

6 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.

6.1 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[4], 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[5].

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 calculated using the first k samples from each process[6];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 [10] and, as a future work, we plan to experiment with other detectors to evaluate their performance.

6.2 Switching heuristic

Section 4 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

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

[5] Usually, evaluation of link delays is also required by other components (for instance, for fault-detection), and can be implemented using low-level acknowledgments.

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

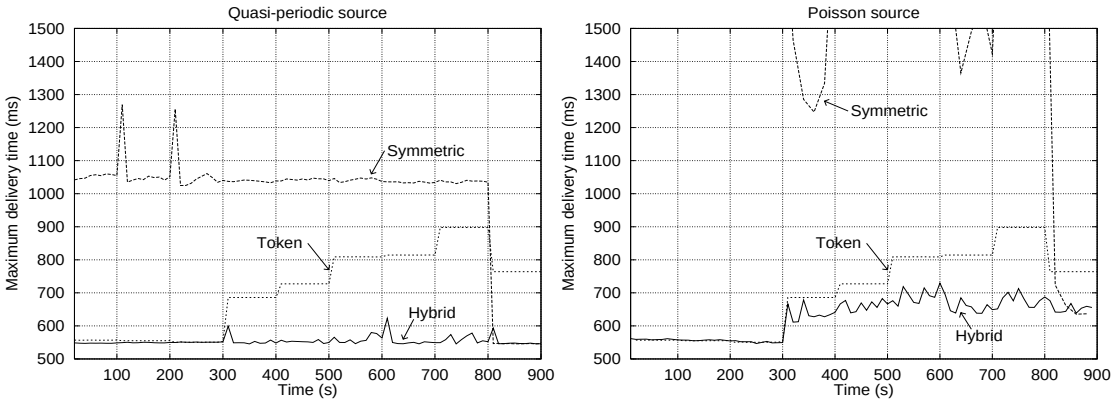


Figure 5: Dynamic hybrid protocol

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).

Results obtained using the heuristic above are presented in Figure 5, alongside with results from symmetric and token-site protocols. For clarity, exactly the same scenarios as in Figure 3 were used, except that now the system ran continuously while the process load changed with time, making the system evolve through all nine scenarios in sequence. Also, results are now presented for both a Quasi-Periodic and a Poisson message source. In the hybrid approach, 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[7]) or token-site

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

protocol.

Throughout this paper, the concepts of the hybrid protocol were illustrated with some relatively simple examples. In an extended report [19] simulations results obtained with more elaborate scenarios are presented, showing that the hybrid protocol offers significant advantages over symmetric or token-site approaches, in large-scale systems and heterogeneous environments.

7 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. 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.

It has been shown that total order protocols can be combined in a hierarchical manner [1]. An optimized solution for topologies based on interconnected LANs could use a hierarchical combination of protocols specially designed for local area networks (such as AMp [24] / xAMp [20] or Totem [2]) with the dynamic hybrid protocol presented here.

Acknowledgments

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

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 International Symposium on Fault-Tolerant Computing Systems*, pages 76–84. IEEE, 1993.

[2] 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. IEEE, 1993.

[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. *Information and Computation*, 118(1):158–179, 1995.

[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. IEEE, 1993.

[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*, pages 296–306. IEEE, 1995.

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

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

[16] 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.

[17] 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. IEEE, 1992.

[18] 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.

[19] L. Rodrigues, H. Fonseca, and P. Veríssimo. Using synchronized clocks for total order in asynchronous systems. Technical report, INESC / IST, 1995.

[20] 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. IEEE, 1992.

[21] 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. IEEE, 1993.

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

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

[24] P. Veríssimo, L. Rodrigues, and M. Baptista. AMp: A highly parallel atomic multicast protocol. In *Proceedings of the SIGCOM'89 Symposium*, pages 83–93. ACM, 1989.