Semi-Passive Replication*

Xavier Défago André Schiper Nicole Sergent
Département d'Informatique
École Polytechnique Fédérale de Lausanne, Switzerland
{*defago,schiper,sergent*}*@lse.epfl.ch*

Abstract

This paper presents the semi-passive replication technique – a variant of passive replication – that can be implemented in the asynchronous system model without requiring a membership service to agree on a primary.

Passive replication is a popular replication technique since it can tolerate non-deterministic servers (e.g., multi-threaded servers) and uses little processing power when compared to other replication techniques. However, passive replication suffers from a high reconfiguration cost in case of the failure of the primary. The semi-passive replication technique presented in the paper benefits from the same advantages as passive replication. However, since it does not require a group membership service, semi-passive replication has a considerably lower cost in case of failure. As explained in the paper, this technique can benefit from an aggressive time-out value significantly lower than what a group membership allows. As a result, the reaction to crashes is greatly improved.

The semi-passive replication algorithm uses failure detectors. The algorithm given in the paper is analysed in the failure free case and in the case of one server crash. The response time (for the client) of these two scenarios is analysed through simulation.

1. Introduction

The effort of building fault-tolerant distributed systems has started a long time ago. A major problem that is inherent to distributed systems is their potential vulnerability to failures. Indeed, whenever a single node crashes, availability of the whole system may be compromised. However, the distributed nature of those systems provides the mean to *increase* the reliability of the system. The distribution makes it possible to introduce redundancy and, thus, to make the overall system more reliable than its individual parts.

1.1. Active and passive replication

Redundancy is usually introduced by the replication of components, or services. Although replication is an intuitive, readily understood idea, its implementation is difficult. Replicating a service in a distributed system requires that each replica of the service keeps a consistent state. This consistency is ensured by a specific replication protocol [1]. Two major classes of replication techniques ensure this: *active* and *passive* replication.

Active replication. In the active replication technique, also called the *state-machine* approach [2], every replica handles the requests received from the client, and sends a reply. In other words, the replicas behave independently and the technique consists in ensuring that all replicas receive the requests in the same order. This technique is appreciated because the response time is low, even in the case of a crash. It has however two important drawbacks: (1) the redundancy of processing implies a high resource usage, and more importantly (2) the requests have to be handled in a deterministic manner. *Determinism* means that the result of an operation depends only on a replica's initial state and the sequence of operations it has already performed. Multi-threaded servers typically lead to non-determinism.

Passive replication. With the passive replication technique, also called *primary-backup* [3, 1], one of the replica, called the *primary*, plays a special role: it receives the requests from the clients and returns responses. The backups interact with the primary only, and receive state update messages from the primary. This replication technique is quite useful since it requires less processing power than active replication and makes no assumption on the determinism of processing a request. However, the implementation of passive replication requires a mechanism to agree on the primary (e.g., a group membership service). If the primary fails, one of the backups takes over. If the primary

*Research supported by OFES under contract number 96.0454, as part of the ESPRIT Project OpenDREAMS-II (number 25262).

crashes before sending a response to the client, the client will eventually time-out. Then, it must learn the identity of the new primary and reissue its request. This leads to a significantly increased response time in the case of failure that makes it unsuitable in the context of time-critical applications. Moreover, passive replication does not totally mask failures to the client.

Passive replication, membership and the cost of an incorrect failure suspicion. Membership is a useful abstraction that constitutes the basic component of many existing group communication systems (e.g., Isis [4], Horus [4, 5], Consul [6], Transis [7], Totem [8]) based on the asynchronous system model. A membership service is used to maintain the list of members of the groups in the system: the service handles *explicit* process' requests to join a group and *explicit* requests to leave a group. The membership service also removes members that are *suspected* to have crashed. This last point leads to an important question: what time-out value should be chosen to exclude members? The trade-off is the following:

- Choosing an aggressive time-out value is costly. Indeed, the cost of the erroneous removal of a correct process from a group is high, as it involves two costly operations: (1) the removal of the process from its group, and shortly after (2) the reintegration of the process in the group.
- Choosing a conservative time-out value is also a bad idea. It can lead to a black-out period of the system (the protocol that uses the input from the membership service is blocked until the crashed process is removed from the group) that is unacceptable for time-critical applications [9].

It is possible to escape from this dilemma by considering two distinct time-out values in the system: (1) one aggressive time-out value (in the order of the round-trip time of a message, i.e., a few milliseconds in a LAN), and (2) one conservative time-out value (many orders of magnitude larger than the aggressive time-out value, e.g., 10 minutes). The aggressive time-out value is just used to suspect crashed processes (without excluding them from the group), while the conservative time-out value can be used to exclude a process from a group.

1.2. Semi-passive replication: passive replication without membership to select the primary

In semi-passive replication, the selection of the primary is not based on a membership service: it is based on the rotating coordinator paradigm that has been used for solving the consensus problem [10]. This allows for the two-timer approach. An aggressive time-out value can be used to ensure fast reaction to the crash of the primary, but incorrect suspicion of the primary due to the aggressive time-out value does not lead the primary to be excluded from the group of servers. If needed, exclusion of a process from the group of servers can be done using a conservative time-out value: this is however out of the scope of the paper. To sum up, semi-passive replication ensures fast reaction to process crash without having a high cost in the case of an incorrect failure suspicion.

The rest of the paper is structured as follows. Section 2 recalls the various replication techniques, and gives an overview of semi-passive replication. Section 3 describes our system model. Section 4 introduces the $\mathcal{DIV}$consensus algorithm, that is used in the semi-passive replication algorithm. In Section 5, we present the semi-passive replication algorithm. In Section 6, we give simulation results for different scenarios. Section 7 concludes the paper.

2. Overview of Semi-Passive Replication

As the name indicates, the semi-passive replication technique has similarities to the passive replication technique. In both techniques there is one single server (called *primary* in passive replication) that (1) processes a request received from a client, and (2) updates the backups after the processing of each request [3]. Because a request is handled by one single server, the processing of the requests can be non-deterministic.

Difference with passive replication. With semi-passive replication, the selection of the server processing the client requests is based on the rotating coordinator paradigm. This can best be described as a variation of a consensus problem among the servers, in which (1) the server processes do not know their initial values when starting the consensus problem (called below *"Consensus with Deferred Initial Values"*, see Section 4), and (2) the decision of the consensus is the *state-update* value obtained by processing the client request. With semi-passive replication, the following characteristics of passive replication are retained:

1. in the absence of server crash or failure suspicion, the request is handled by one single server, and
2. the processing of a request does not need to be deterministic.

A minor difference between passive and semi-passive replication is that, with semi-passive replication, the client sends its request to all replicas and every replica sends a response back to the client. So there is no need for the client to know the identity of the primary, nor for the client to have time-outs to detect the crash of the primary (and reissue the request): similarly to active replication, with semi-passive replication, the effect of failures are completely masked to the client.

Difference with semi-active replication. The semi-active replication technique [11] was introduced to circumvent the problem of non-determinism with active replication, in the context of time-critical applications. This technique is based on active replication and extended with the notion of *leader* and *followers*. While the actual processing of a request is performed by all replicas, it is the responsibility of the leader to perform the non-deterministic parts of the processing and inform the followers. This technique is close to active replication, with the difference that non-deterministic processing is possible.

3. System model

3.1. Asynchronous system with failure detectors

Semi-passive replication is based exclusively on failure detectors, and does not require any membership service. This means that the semi-passive replication technique can be described without reference to process recovery, i.e., the semi-passive replication algorithm can be described in the system model of [10].

In the paper we consider the asynchronous system model augmented with failure detectors (The asynchronous system model has to be augmented with failure detectors to make agreement problems (e.g., consensus) solvable [12, 10]). We assume only crash failures of processes (no Byzantine faults). As in [10], we assume that channels are reliable, and that processes do not recover after a crash. The model is particularly attractive due to its simplicity, by incorporating the minimal assumptions needed to solve agreement problems (e.g., consensus). *However, these assumptions are only made to avoid a discussion that is not central to the contribution of this paper*. Indeed, the model can be extended to lossy channels and network partitions [13], and to process recovery after a crash [14].

3.2. Clients and servers

We consider one single replicated service. The processes that implement this service are called *server processes*, and the processes sending invocations to the replicated service are called *client processes*. The set of server processes is denoted by Ω_S, while the set of client processes is denoted by Ω_C. We assume a majority of correct processes in Ω_S. The number of processes in the set of servers is denoted by $n = |\Omega_S|$.

3.3. Failure detectors

We consider the $\Diamond\mathcal{S}$ failure detector on the set of server processes Ω_S: $\Diamond\mathcal{S}$ is the weakest failure detector for solving consensus [15], and is defined by the following properties [10]:

Strong Completeness. There is a time after which every process in Ω_S that crashes is permanently suspected by all correct processes in Ω_S.

Eventual Weak Accuracy. There is a time after which some correct process in Ω_S is never suspected by any correct process in Ω_S.

4. Consensus with Deferred Initial Values

This section introduces the definition of Consensus with Deferred Initial Values, which will be used in the semi-passive replication algorithm (Sect. 5).

The problem. In the classical definition of the consensus problem, every process p_i starts with an initial value v_i, and the processes have to agree on a common decision value. In the $\mathcal{DIV}$consensus ($\mathcal{DIV}$ stands for Deferred Initial Values) a process p_i *is not required to have its initial value defined when starting the consensus algorithm*. Hence the algorithm can request an initial value from p_i at a later time.

Getting initial values. The interface of a consensus algorithm is usually expressed as a function $propose(v_i)$, where the argument v_i is the initial value of the process calling the $propose$ function. This is not suitable for consensus with deferred initial values. Instead, we propose to express the interface of the $\mathcal{DIV}$consensus algorithm as a function $DIVpropose(getInitVal)$, with the function *getInitVal* as argument. The function *getInitVal* returns an initial value, and is called during the execution of the consensus algorithm, when an initial value is needed.

When solving $\mathcal{DIV}$consensus using the rotating coordinator paradigm [1] [10], the function *getInitVal* may be called by some process p_i only when p_i is coordinator. So, in the absence of failures, only one process calls the *getInitVal* function. We show in Section 5 how this is exploited in the semi-passive replication algorithm.

Solving $\mathcal{DIV}$consensus. The $\mathcal{DIV}$consensus problem can easily be solved by small changes to the Chandra-Toueg consensus algorithm based on the $\Diamond\mathcal{S}$ failure detector [10]. The algorithm requires a majority of correct processes, but understanding its details is not essential. In good runs only the coordinator of the first round (process p_1) calls the *getInitVal* function, and the $\mathcal{DIV}$consensus problem is solved in one round (Fig. 1). With one crash (and no incorrect suspicions), two rounds are needed to solve the problem (see [16]): Process p_1, coordinator of the first round, calls

[1] In the rotating coordinator paradigm, the algorithm goes through a succession of rounds, and in every round another process is coordinator.

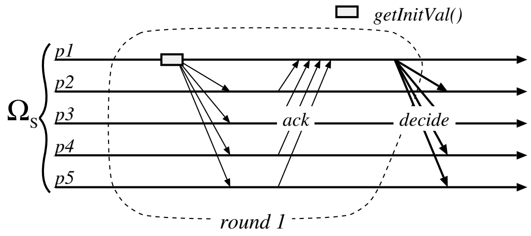

Figure 1. $\mathcal{DIV}$consensus with $\diamond\mathcal{S}$ (good run)

getInitVal and crashes. The surviving processes move to the second round. Process p_2, coordinator of the second round, in turn calls *getInitVal*.

5. Semi-Passive Replication

5.1. A sequence of $\mathcal{DIV}$consensus

We begin this section by giving a general overview of the solution. We then introduce our notation and present the semi-passive replication technique expressed as a sequence of $\mathcal{DIV}$consensus. Then, in order to give a feeling of the messages generated by the solution, we consider two scenarios: (1) no failure, and (2) the worst case with the crash of one replica and no incorrect suspicion. The "no failure" case is close to the communication schema of the passive replication technique, and should look familiar to the reader.

Overview of the solution. Similarly to passive replication, in the semi-passive replication technique the primary handles the requests and, after the processing of each request, sends an *update* message to the backups. In our solution based on a sequence of consensus problems, every consensus problem *decides on the content of the update message*. This requires the initial value of every consensus problem to be an *update value*, and explains the need for consensus with deferred initial values. Consider the consensus problem number k. *As the initial value for consensus k is an update value, a server process can have an initial value for consensus k only after having handled a new request received from a client*. However, only the primary handles client requests in the semi-passive replication technique, i.e., only the primary has an initial value for consensus k. This poses no problem as long as the primary does not crash (or is not suspected to have crashed): the primary imposes its initial value as the decision value (and so, the backups do not need to have an initial value for the consensus k.). However, if the primary crashes, then one of the backups needs to handle the request to obtain an initial value for consensus number k. This processing takes place *after the start of consensus number k*, and explains the need for deferred initial values in the context of the consensus problem.

1: Initialisation:
2: $\quad recv_s \leftarrow \emptyset$ $\quad$ {*hold the received requests*}
3: $\quad hand_s \leftarrow \emptyset$ $\quad$ {*hold the handled requests*}
4: $\quad k_s \leftarrow 0$

5: **when** *receive*(req_c^j) from client
6: $\quad recv_s \leftarrow recv_s \cup \{req_c^j\}$

7: **function** *getInitVal* ()
8: $\quad req_{c\,s} \leftarrow$ [select one request in $(recv_s - hand_s)$]
9: $\quad (upd_s^{k_s}, res_s^{k_s}) \leftarrow handle_s(req_{c\,s}, state_s^{k_s-1})$
10: $\quad$ **return** $(req_{c\,s}, upd_s^{k_s}, res_s^{k_s})$

11: **when** $recv_s - hand_s \neq \emptyset$
12: $\quad k_s \leftarrow k_s + 1$ $\quad$ {*Solve the $k_s{}^{th}$ $\mathcal{DIV}$consensus* }
13: $\quad (req_c, upd_{s'}^{k_s}, res_{s'}^{k_s}) \leftarrow$ ***DIVpropose***(k_s,*getInitVal*)
14: $\quad$ send $(res_{s'}^{k_s})$ to c
15: $\quad state_s^{k_s} \leftarrow apply(upd_{s'}^{k_s}, state_s^{k_s-1})$ $\quad$ {*Update the state*}
16: $\quad hand_s \leftarrow hand_s \cup \{req_c\}$

Figure 2. Semi-passive replication (server s)

In other words, expressing semi-passive replication as a sequence of $\mathcal{DIV}$consensus hides the role of the primary inside the consensus algorithm. A process p takes the role of the primary (i.e., handles client requests) exactly when p is the coordinator in the consensus algorithm. So, in every instance of the consensus problem, the mechanism for selecting a coordinator is the mechanism that selects the primary.

The notation. In order to express the algorithm, we introduce the following notation.

- req_c^j: j^{th} request sent by a client c.
- upd_s^i: update message number i, as generated by server s.
- res_s^i: response number i for the client, as generated by server s.
- $state_s^i$: state of server s after the processing of upd_s^i.
- $handle_s : (req_c^j, state_s^{i-1}) \longmapsto (upd_s^i, res_s^i)$
  Transformation of a request req_c^j into an update message (handling the request).
- $apply : (upd_{s'}^i, state_s^{i-1}) \longmapsto state_s^i$
  Modification of the state due to the application of the update message $upd_{s'}^i$ (possibly $s = s'$).

The full algorithm. We give now the full algorithm, which expresses the semi-passive replication technique as a sequence of $\mathcal{DIV}$consensus.

The algorithm, executed by every server process, is given in Figure 2. Every server s manages an integer k_s (line 4), which identifies the current instance of the consensus problem. Every server process also handles the variables $recv_s$ and $hand_s$ (lines 2,3):

- $recv_s$ is a set containing the requests received by s from the clients. Whenever s receives a new request, it is included in $recv_s$ (lines 5,6);
- $hand_s$ is a set which consists of the requests that have been handled (not necessarily by s). A new consensus is started by s whenever the preceding consensus is terminated, and $recv_s - hand_s$ is not empty (line 11). At the end of the consensus, the request that has been handled is inserted by s into $hand_s$ (line 16).

The function *getInitVal* is discussed in the next paragraph. The main part of the algorithm consists in lines 11–16: at line 13 the $\mathcal{DIV}$consensus function is called, which returns the decision $(req_c, upd_{s'}^{k_s}, res_{s'}^{k_s})$: req_c is the request that has been handled. $upd_{s'}^{k_s}$ is the update resulting from s' handling req_c, and $res_{s'}^{k_s}$ is the response that should be sent to the client c. At line 14 the response $res_{s'}^{k_s}$ is sent to the client. At line 15 the local state of the server s is updated according to the update message $upd_{s'}^{k_s}$. Finally, at line 16 the request that has been handled is inserted into the set $hand_s$.

Let us now come to the discussion of the function *getInitVal* that returns initial values for each instance of the consensus problem. The function *getInitVal* (line 7) is called from the $\mathcal{DIV}$consensus algorithm whenever a process is coordinator, i.e., acts as the primary in the context of the semi-passive replication technique. The function selects a client request that has not been handled yet (line 8), handles the request (line 9), and returns the selected request $req_{c\,s}$, the update message $upd_s^{k_s}$ resulting from handling the request, as well as the corresponding response message $res_s^{k_s}$ (line 10).

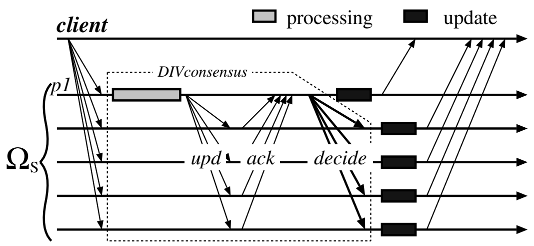

Figure 3. Semi-passive replication (good run)

The solution in good runs. We call "good run" a run in which no server process crashes and no failure suspicion is generated. In good runs, our general solution leads to a schema similar to the well known communication schema of passive replication (see Fig. 3). Let Figure 3 represent the execution of consensus number k. The server process p_1 is the initial coordinator for consensus k and also the primary. After having received the request from the client, the primary p_1 handles the request. Once the processing is done, p_1 has the initial value for consensus k. According to the consensus protocol, p_1 multicasts the update message upd to the backups, and waits for ack messages. Once ack messages have been received (actually from a majority), process p_1 can decide on upd, and multicast the $decide$ message to the backups. As soon as the $decide$ message is received, the servers update their state, and send the reply to the client.

We can see from Figure 3 that the latency for the client, expressed in number of communication steps, is equal to 4: (1) the request sent by the client to the servers Ω_S, (2) the update message sent by the primary to the backups, (3) the ack messages sent by the backups to the primary, and finally (4) the reply sent by the primary to the client. Notice that the reply sent by a backup to the client is received after 5 communication steps: as the client waits only for the first reply, the latency of the client corresponds indeed to 4 communication steps.

The solution in the case of one crash. Apart from the "good run" scenario, we also evaluate the worst case latency for the client in the case of one crash. In this evaluation, we consider that there is no incorrect suspicion. As illustrated in Figure 4, the worst case scenario happens when the primary p_1 (i.e., the initial coordinator of the consensus algorithm) crashes immediately after processing the client request, but before being able to send the update message upd to the backups (compare with Fig. 3).

In this case, the execution of the consensus algorithm is as follows. If the primary p_1 crashes, the backups eventually suspect p_1, send a negative acknowledgement message $nack$ to p_1 (the message is needed by the consensus algorithm), and start a new round. The server process p_2 becomes the coordinator for the new round, i.e., becomes the new primary, and waits for $estimate$ messages from a majority of servers: these messages might contain an initial value for the consensus, in which case p_2 needs not to process the client request again. In our worst case scenario, the initial primary p_1 has crashed before being able to multicast the update value upd. So none of the $estimate$ messages received by p_2 contains an initial value. In order to obtain one, the new primary p_2 processes the request received from the client (Fig. 4), and from that point on, the scenario is similar to the "good run" case of the previous section (compare with Fig. 3).

As shown in Figure 4, there are five communication steps between the emission of the request by the client, and the reception of the first reply (the $nack$ messages in Figure 4 are not on the critical path and do not increase the number

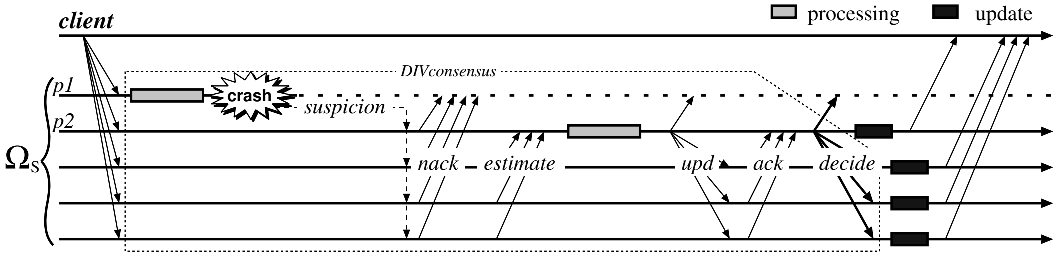

Figure 4. Semi-passive replication with one failure (worst case).

of communication steps). This is one communication step more than in the good run scenario. In addition, the latency is increased by two other factors. The first factor is the time needed by the backups to suspect the primary. The second factor is the processing time of the request: as shown in Figure 4, the client request is handled twice in the worst case scenario, once by the old primary and once by the new primary. We come back to these issues in Section 6, where we compute the latency by simulation.

5.2. Improving the failure resilience

The reader may have noticed, from Figure 4, the following problem with our semi-passive replication algorithm. In the rotating coordinator paradigm, every instance of the consensus (or $\mathcal{DIV}$consensus) algorithm invariably starts the first round by selecting the same process, say p_1, as the coordinator. If p_1 crashes, further executions of the consensus algorithm will always require at least two rounds to complete. This extra cost (two rounds instead of one) can be avoided by a very simple idea.

Assume that, for the consensus number k, the processes of Ω_S are ordered $[p_1, p_2, p_3, p_4, p_5]$, which defines p_1 as the first coordinator. If p_1 has crashed (i.e., is suspected) during consensus k, the processes of Ω_S are reordered $[p_2, p_3, p_4, p_5, p_1]$ for the consensus $k+1$, which defines p_2 as the first coordinator. So, despite the crash of p_1 in consensus k, consensus $k+1$ can be solved in one single round. This reordering can be implemented in the context of the $\mathcal{DIV}$consensus algorithm, with no additional messages. The general idea is to modify the algorithm to have the decision value made on a pair $(initValue, sysList)$, where *initValue* is the decision value considered so far, and *sysList* is a permutation of Ω_S. In every round, the coordinator reorders the *sysList* so that the most frequently suspected processes appear last.

6. Simulation

In this section, we evaluate the performance of semi-passive replication, using discrete event simulation. This choice is motivated by two main reasons. First, simulation allows us to get results faster since it does not need a full-fledged implementation. Second, simulation allows a fair comparison of the performance of the different scenarios, since they are obtained using identical assumptions and simulation conditions (this fairness would be impossible to achieve in a real system, where the measurements are biased by unpredictable network and station loads).

6.1. Assumptions

We consider processes (client and server processes) communicating using a datagram transport service (UDP/IP). We simulate both point-to-point communication and broadcast communication (IP multicast). The network modelled is a multi-access LAN (Ethernet). The network controller has its own processor which is responsible for implementing the Ethernet protocol. We make the following assumptions:

- The workstations connected to the network are identical and uniformly distributed along the physical medium.
- There is only one process (client or server process) running on each workstation.
- The LAN is private, i.e., there is no other message passing on the network except those generated by the replication protocol under study.
- We consider a message loss rate of 10^{-6} (it includes message collisions and transmission errors).
  In-house observations and studies in Ethernet networks show that the actual loss of messages is about one in two millions [17] whenever the network traffic is far below the achievable throughput [18]. This assumption is fulfilled in the context of our simulations, where the generated load is under 15% of the available bandwidth (this can be verified by computing the number of messages generated during the simulation).

6.2. Model for UDP and Ethernet

In some simulation studies using discrete event simulation, where the transmission delay between two processes is assumed to follow a particular empirical distribution. Unlike this approach, we consider a finer-grained model [19]. A message m sent from process p running on workstation WS_p to process q running on workstation WS_q requires the allocation of three resources:

- the CPU on WS_p, to execute the UDP/IP send;
- the network;
- the CPU on WS_q, to execute the UDP/IP receive;

Whenever one of these resources is busy when needed by a message m, then m has to wait. CPU resources are allocated based on a FIFO policy. The network resource is allocated locally based on a FIFO policy, and randomly between workstations. We consider the following constants:

- sending a message requires the CPU of the sending workstation for t_S time units;
- the network resource is needed for t_N time units;
- receiving a message requires the CPU of the receiving workstation for t_R time units.

A similar communication model may be found in [20]. In the simulation, we have considered the following values for each message frame (messages of 1500 bytes): $t_S = 269\mu$s, $t_N = 107\mu$s and $t_R = 292\mu$s. Validation of this model is described in [21].

6.3. Simulation results

We have simulated the semi-passive replication algorithm presented in Section 5.1, in the context of the two scenarios presented in that section: good run and one crash. In the simulation, we have considered the following values:

- time needed to process a client request: 1 ms.
- size of the messages (except "update"): 1 kB.
- size of "update" messages: 1 kB, 4 kB, 16 kB, 64 kB.
- time needed to update the server state: 20 μs/kB.
- copy-on-write time:[2] 20 μs/kB.

We have run simulations for the cases of 3, 5 and 7 servers, and measured the response time (time between the sending of the request by the client and the reception of the response) in good runs (no crashes, no failure suspicion) and in the worst case of one crash (the crash of the coordinator of the first round in the consensus, no incorrect suspicion). The results are given in Table 1, both for the case of point-to-point communication and for the case of broadcast communication (IP multicast).

n	size of "update" msg [kB]	point-to-point [ms] time-out	point-to-point latency good run	point-to-point latency one crash	diffusion [ms] time-out	diffusion latency good run	diffusion latency one crash
3	1	10	5.6	18.1	10	5.3	16.9
	4	15	18.3	32.3	10	14.8	23.7
	16	41	68.8	109.6	25	56.3	80.8
	64	158	281	438.2	95	243.1	339.3
5	1	10	6.1	19.6	10	5.9	17.9
	4	22	30.1	55.1	10	15.4	26.8
	16	72	129.6	203.4	25	57.8	83.9
	64	283	524.9	810.3	95	252.1	349.5
7	1	10	6.7	20.7	10	6.6	18.6
	4	29	38.2	70.8	10	16	28.4
	16	102	178.6	284.4	25	58.8	86.1
	64	402	724.7	1130	95	261.1	358.6

Table 1. The client latency - simulation results (n=number of server processes)

6.4. Discussion

The time-out values of Table 1 have been chosen to avoid the generation of suspicions in runs with no crash, and to minimize the time for detecting the crash of a process. These optimum time-out values have been determined using simulation, by successively increasing the time-outs in case of incorrect suspicions. We can notice in Table 1 that the optimum time-outs depend on the number of processes, the size of the update messages and the type of network (point-to-point or diffusion). The reason is that each of these three factors have a direct influence on the traffic on the network and thus, on the end-to-end delay of messages.

It can also be noticed from Table 1 that the latency of the semi-passive replication algorithm in case of one crash is approximately the latency of good runs plus the time-out value. This result could have been predicted by comparing the communication schemas of Figure 3 and Figure 4. In runs with one process crash, the latency is increased (compared to good runs) at least by the time-out delay needed for detecting the crash.

Not surprisingly, the best latency (5.6 ms, 18.1 ms) is obtained for 3 processes, update messages of 1 kB, diffusion network, and the worst latency (724 ms, 1130 ms) for 7 processes, update messages of 64 kB, and point-to-point communication network.

[2] We consider the following implementation. Every attempt to modify the state by the primary is trapped, and a copy of the memory page is taken (copy-on-write). After the processing of the request, the modified pages constitute the update message.

7. Conclusion

This paper has introduced a new replication strategy, called semi-passive replication. The technique has similarities with passive replication, but does not require an agreement problem to be solved (e.g., membership) to select the "primary". This reduces the latency of the client in the case of the crash of the "primary". It also reduces the cost of incorrectly suspecting the "primary", because in semi-passive replication there is no need for an incorrectly suspected process to "rejoin" (as there is no "exclusion" of suspected processes, there is no need for "joins").

For these reasons, semi-passive replication allows us to select aggressive time-outs, which reduce the time needed to react to the crash of the "primary". This allows the "two-timer" approach that we advocate: an aggressive time-out value (in the order of milliseconds) to ensure fast reaction to crashes, and a conservative time-out value (in the order of seconds or minutes) to exclude a process from the group of servers. This two-timer approach was already possible for active replication – a technique requiring an atomic multicast communication primitive – because the atomic multicast algorithm given in [10] does not require a membership. We believe that the two-timer approach has clear advantages for time-critical applications, but none of the existing systems exploit these advantages, because of their heavy use of a membership service.

By extending the applicability of the two-timer approach to the class of "passive" replication techniques – in which only one server handles the client request – we hope that the advantages of this approach will finally be understood and become more popular. We can mention that the semi-passive replication technique has been successfully implemented in the context of a CORBA Object Group Service [22].

References

[1] R. Guerraoui and A. Schiper, "Fault-tolerance: from replication techniques to group communication," *IEEE Computer*, vol. 30, pp. 68–74, Apr. 1997.

[2] F. Schneider, "Replication management using the state-machine approach," in *Distributed Systems* (S. Mullender, ed.), ch. 7, pp. 169–198, Addison-Wesley, second ed., 1993.

[3] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," in *Distributed Systems* (S. Mullender, ed.), ch. 8, pp. 199–216, Addison-Wesley, second ed., 1993.

[4] K. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1993.

[5] R. van Renesse, K. P. Birman, B. B. Glade, K. Guo, *et al.*, "Horus: A flexible group communications system," TR 95-1500, Cornell University, Mar. 1995.

[6] S. Mishra, L. L. Peterson, and R. D. Schlichting, "Consul: A communication substrate for fault-tolerant distributed programs," *Distributed Systems Engineering Journal*, vol. 1, no. 2, pp. 87–103, 1993.

[7] Y. Amir, D. Dolev, S. Kramer, and D. Malki, "Transis: a communication sub-system for high availability," in *Proc. of the 22nd Int. Symp. on Fault-Tolerant Computing*, July 1992.

[8] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulis, and T. P. Archambault, "The Totem system," in *Proc. of the 25rd Int. Symp. on Fault-Tolerant Computing*, pp. 61–66, 1995.

[9] R. Guerraoui and A. Schiper, "Consensus: the big misunderstanding," in *Proc. of the 6th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems*, pp. 183–188, IEEE Computer Society Press, Oct. 1997.

[10] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, pp. 225–267, Mar. 1996.

[11] D. Powell, *Delta4: A Generic Architecture for Dependable Distributed Computing*, vol. 1 of *ESPRIT Research Reports*. Springer-Verlag, 1991.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, pp. 374–382, Apr. 1985.

[13] M. K. Aguilera, W. Chen, and S. Toueg, "Quiescent reliable communication and quiescent consensus in partitionable networks," TR 97-1632, Cornell University, June 1997.

[14] R. Oliveira, R. Guerraoui, and A. Schiper, "Consensus in the crash-recover model," TR 97/239, École Polytechnique Fédérale de Lausanne, Switzerland, Aug. 1997.

[15] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *J. ACM*, vol. 43, pp. 685–722, July 1996.

[16] X. Défago, A. Schiper, and N. Sergent, "Semi-passive replication," TR 98/277, École Polytechnique Fédérale de Lausanne, Switzerland, May 1998.

[17] L. F. Cabrera, E. Hunter, M. Karels, and D. Mosher, "User-process communication performances in networks of computers," *IEEE Trans. Software Engineering*, vol. 14, Jan. 1988.

[18] M. Santifaller, *TCP/IP and ONC/NFS- Internetworking in a UNIX Environment*. Addison-Wesley, 1994.

[19] N. Sergent, "Performance evaluation of a consensus algorithm with Petri nets," in *Proc. of the 7th Int. Conf. on Petri Nets and Performance Models*, June 1997.

[20] N. Malcolm and W. Zhao, "Hard real time communication in multiple-access networks," *Real-Time Systems*, vol. 8, pp. 35–77, 1995.

[21] N. Sergent, "Evaluating latency of distributed algorithms using Petri nets," in *Proc. of the 5th Euromicro Workshop on Parallel and Distributed Processing*, Jan. 1997.

[22] P. Felber, R. Guerraoui, and A. Schiper, "The implementation of a CORBA object group service," *Theory and Practice of Object Systems*, vol. 4, no. 2, pp. 93–105, 1998.