Design and Implementation of a Consistent Time Service for Fault-Tolerant Distributed Systems [*†]

W. Zhao
Department of Electrical and Computer Engineering
Cleveland State University, Cleveland, OH 44145
wenbing@ieee.org

L. E. Moser and P. M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
moser@ece.ucsb.edu, pmms@ece.ucsb.edu

Abstract

Clock-related operations are one of the many sources of replica non-determinism and of replica inconsistency in fault-tolerant distributed systems. In passive replication, if the primary server crashes, the next clock value returned by the new primary server might have actually rolled back in time, which can lead to undesirable consequences for the replicated application. The same problem can happen for active replication when the result of the first replica to respond is taken as the next clock value.

In this paper we describe the design and implementation of a Consistent Time Service for fault-tolerant distributed systems. The Consistent Time Service introduces a group clock that is consistent across the replicas and that ensures the determinism of the replicas with respect to clock-related operations. The group clock is monotonically increasing, is transparent to the application, and is fault-tolerant. The Consistent Time Service guarantees the consistency of the group clock even when faults occur, when new replicas are added into the group, and when failed replicas recover.

Keywords: Replica Consistency, Clock Synchronization, Replica Non-Determinism, Fault Tolerance

[*]An earlier version of this paper was presented at the IEEE International Conference on Dependable Systems and Networks, San Francisco, California, 2003. This research was supported by DARPA/ONR Contract N66001-00-1-8931 and MURI/AFOSR Contract F49620-00-1-0330.

[†]**To appear in Computer Systems Science and Engineering Journal**

1 Introduction

One of the biggest challenges of replication-based fault tolerance is maintaining replica consistency in the presence of replica non-determinism [14]. For active replication, it has been recognized that the replicas of a process, object or other component must be deterministic, or rendered deterministic. Consequently, passive replication, based on the primary/backup approach, has been advocated if the potential for replica non-determinism exists; however, the same replica non-determinism problems that arise for active replication during normal operation arise for passive replication when the primary replica fails.

Clock-related operations, such as `gettimeofday()`, are one source of replica non-determinism. Clock-related operations are common in many distributed systems applications such as transaction processing, stock trading, and authentication systems.

Most general-purpose and soft real-time distributed applications utilize an event-triggered approach. For such applications, software-based clock synchronization algorithms [9, 15, 16, 19] cannot solve the replica non-determinism problem for clock-related operations. The primary reason is that the replicas in a group of replicas can read different clock values when they process the same request at different real times due to asynchrony in processing and/or scheduling at the replicas. Even if different replicas carry out the same clock read operation at the same real time, traditional clock synchronization algorithms cannot ensure strong replica consistency of the clock values, because they cannot guarantee perfect precision of the hardware clocks of different processors.

To guarantee replica consistency in the presence of clock-related non-determinism, fault-tolerant systems such as Mars [8] have used a lock-step, time-triggered approach. However, the time-triggered approach is not applicable in all circumstances, due to its requirement of *a priori* scheduling of the operations of the replicated application. In particular, the application program cannot read the clock time because no mechanism to ensure consistency of the readings of the clocks is provided.

In [11] a pre-processing approach is proposed to render deterministic the computations of the replicas. The pre-processing involves running a distributed consensus protocol to harmonize the inputs from the environment. The primary/backup approach is used to produce deterministic readings of the clocks for the group of replicas. The physical hardware clock value of the primary is returned, and the primary conveys the result to all of the backups. The other replicas subsequently use that clock value, instead of their own physical hardware clock values.

The primary/backup approach differs from our approach, particularly for active replication, where there are no primary or backup replicas and where all of the replicas supply proposed consistent clock values and compete to provide the consistent group clock value.

Although the primary/backup approach solves the consensus problem for individual clock readings for the replicas in the group, it does not guarantee that the clock readings always advance forward. If the primary crashes, the newly selected primary starts with its own physical hardware clock value for the next clock reading. Because of the differences in the physical hardware clocks of the old and the new primary replicas, and the gap in time of the computation for the two replicas, the next clock reading might be earlier than the previous clock reading before the primary crashed. Roll-back of the clock can break the causal relationships between events in the distributed system, and can lead to undesirable consequences for the replicated application [18].

It might also happen that two consecutive clock readings from two different replicas (due to the failure of the original replica) differ too much in the other direction, *i.e.*, the second clock reading is too far ahead of the first clock reading. The presence of this fast-forward behavior can lead to unnecessary time-outs in the replicated application.

The clock roll-back and fast-forward problems associated with the primary/backup approach can be alleviated by closely synchronizing the physical hardware clocks. Clocks can be synchronized in a fairly accurate manner using the Network Time Protocol (NTP) [10] or Global Positioning Satellite (GPS) clocks.

The Hypervisor fault tolerance system [4] uses an improved primary/backup approach by building a consistent state machine for a group of replicas. A backup replica synchronizes with the primary replica periodically for every predetermined number of instructions (called an epoch) executed by the Hypervisor virtual machine. The primary replica sends the results of all non-deterministic operations, including clock values, to the backup at the end of each epoch. The backup replica stores the clock value of the last epoch. If the primary replica fails during a new epoch, the backup replica takes over the role of the primary, using the stored clock value as the starting point so that the clock never rolls back. However, the Hypervisor approach requires the ability to count the number of instructions that a processor executes. Such a requirement is becoming harder to satisfy with the increasing asynchrony built into modern processors [6].

In this paper we present the design and implementation of a Consistent Time Service for fault-tolerant distributed systems. The Consistent Time Service ensures deterministic clock-related operations for the replicas in a group of replicas, even in the presence of faults.

The Consistent Time Service utilizes a group clock that is consistent across the replicas and that ensures the determinism of the replicas with respect to clock-related operations. The group clock is monotonically increasing, is transparent to the application, and is fault tolerant. The Consistent Time Service allows the addition of new replicas and the recovery of failed replicas, while guaranteeing the consistency of the group clock.

2 Consistent Time Service

The Consistent Time Service applies to active replication and to the primary/backup approach used by passive and semi-active replication. In active replication, all of the replicas are equal (there are no primary or backup replicas), and all of the replicas transmit messages containing requests and replies, receive messages, and process requests and replies. In passive replication, only one of the replicas, the primary replica, processes incoming messages and transmits outgoing messages. The primary replica periodically checkpoints its state and transmits that state to the backup replicas. The backup replicas log incoming messages and the checkpoints from the primary replica.

Semi-active replication is a hybrid replication technique that was introduced in the Delta-4 project [14]. It combines properties of both active replication and passive replication. In semi-active replication, both the primary and the backup replicas process incoming messages. However, the primary replica conveys the results of any non-deterministic operations to the backup replicas, so that they remain consistent with the primary replica.

The Consistent Time Service, described in this paper, is implemented on top of a replication infrastructure and a multicast group communication system [1]. The reliable ordered delivery protocol of the multicast group communication system ensures that the replicas receive the same messages in exactly the same order. The Consistent Time Service employs library interpositioning of clock-related system calls to achieve application transparency. The membership of the group of replicas is managed by the replication infrastructure and the underlying group communication system.

The replicas in the group of replicas are assumed to be fail-stop, as are the physical clocks, *i.e.*, a non-faulty replica never sends a wrong clock value to the other replicas. The fail-stop assumption for physical clocks might appear to be overly restrictive. However, most group communication systems operate only if the physical clocks are fail-stop. Arbitrary fault models for physical clocks can disrupt the timeout-based fault detection strategy that group communication systems use.

Network partitioning faults are handled by the underlying group communication system, which uses a primary component model [3] to handle network partitioning and remerging, *i.e.*, only the primary component survives a network partition.

The Consistent Time Service depends on the consistent clock distribution and synchronization (CCDS) algorithm, described in Section 3. The CCDS algorithm presents a consistent view of the clocks for the replicas in a group of replicas. Hence, the application at each replica sees a *consistent group clock* instead of the inconsistent physical hardware clocks of the replicas.

At least one replica in the group of replicas is assumed to be non-faulty during a round of the CCDS

algorithm (the definition of round is given in Section 3). Communication faults are handled and masked by the underlying group communication system, *i.e.,* the CCDS algorithm assumes a reliable communication channel. If a replica in the group is detected to be faulty, it is removed from the membership of the group. The failure of a replica does not interfere with the execution of the CCDS algorithm.

All threads that perform clock-related operations are created during the initialization of a replica, or during runtime, in the same order at different replicas. Except for timer management, one and only one thread processes incoming remote method invocations, sends nested remote method invocations, and handles the corresponding replies.

3 The CCDS Algorithm

We present the Consistent Clock Distribution and Synchronization (CCDS) algorithm for active replication, and discuss modifications of the algorithm for passive replication and semi-active replication.

The CCDS algorithm proceeds in rounds. A *round* is a period of time in which the mechanisms retrieve the physical hardware clock values, exchange messages, reset the clock offset, and decide on the consistent clock value for the group. A new round of clock distribution and synchronization is started for each clock-related operation.

Within a single thread, all clock-related operations are naturally sequential; a thread cannot start a new round before the current round completes. The scheduling algorithm of the replication infrastructure [2, 7, 12] determines whether or not there are multiple concurrent consistent clock distributions and synchronizations in progress for different threads.

The basic idea of the CCDS algorithm is that the replica in the group, whose message containing a proposed group clock value is ordered and delivered first, decides on the group clock value for that particular round. That replica is the *synchronizer* for the round. Thus, in the CCDS algorithm the replicas in the group compete to become the synchronizer for the round. Each of the replicas keeps an *offset* to maintain the clock readings for the group, even if the synchronizer for different rounds changes. If necessary, the offset is re-adjusted for each round.

In a round, the group clock is set to the local clock proposed for the group clock by the winner (the synchronizer) of that round. In the initial round, the group clock is initialized to the synchronizer's local clock value, which is the value of the physical hardware clock of that replica. In each subsequent round, the group clock is set to the synchronizer's local clock value, which is the physical hardware clock value of that replica plus the offset of the group clock from the local clock of that replica in the previous round.

If the message containing the proposed group clock is delivered to any non-faulty replica, it will be

delivered to all non-faulty replicas. Because at least one replica in the group is non-faulty during a round, the CCDS algorithm determines a consistent group clock value for that round.

For the primary/backup replication, the synchronizer is the primary replica. If the primary replica fails during the round before it sends the consistent clock synchronization message or if it fails during the round after it sends the consistent clock synchronization message, but the message is not delivered to any non-faulty replica, the new primary replica sends a consistent clock synchronization message.

3.1 Data Structures

Messages

The CCDS algorithm requires the sending of a control message, called the *Consistent Clock Distribution and Synchronization (CCDS)* message, to the replicas in the group.

Each message contains a common fault-tolerant protocol message header. The header contains the following fields:

- `msg_type`: The type of the message, *i.e.*, CCDS.
- `src_grp_id`: The source (sending) group identifier.
- `dst_grp_id`: The destination (receiving) group identifier. For a CCDS message, the source group identifier and the destination group identifier are the same.
- `conn_id`: The identifier that uniquely determines a connection that has been established between the source group and the destination group.
- `msg_seq_num`: The sequence number of the message sent on the connection. For a CCDS message, this field contains the CCDS round number.
- `grp_clock`: The group clock value of the last round. This field is used to ensure causality of inter-group interactions.

For a regular user message, the `src_grp_id`, `dst_grp_id` and `conn_id` uniquely determine a connection within the distributed system. The `msg_seq_num` uniquely determines a message within the connection. These fields together constitute the message identifier.

The payload of a CCDS message contains two parts:

- `Sending thread identifier`: The identifier of the sending thread.
- `Proposed group clock`: The sum of the physical hardware clock value and the clock offset at the replica.

Local Data Structures

For each replica, the CCDS algorithm employs the following local data structures:

- `my_physical_clock_val`: The variable that stores the physical hardware clock value read at the beginning of each round.
- `my_clock_offset`: The offset of the group clock value from the physical hardware clock value of the local replica. The offset is set once for each round, and is used to calculate the local clock value that the replica proposes for the group clock in the next round.
- `my_round_number`: The CCDS algorithm round number for the replica. This round number is used to perform duplicate detection and to match the clock-related operation with the corresponding CCDS message in the same round.
- `my_common_input_buffer`: A buffer that queues CCDS messages for a slow replica when the thread that will perform the same logical operation has not yet been created. In this case, a matching `CCDS_handler` (defined below) to process the received CCDS messages does not exist.
- `CCDS_handler`: The consistent clock distribution and synchronization handler object. There is one such handler object for each thread. Each `CCDS_handler` object contains the following member variables and member functions:
  - `my_thread_id`: The identifier of the thread.
  - `my_input_buffer`: The buffer that stores the received CCDS messages sent by the replicas in the same group. Even though all of the clock-related operations in a thread are sequential, slower replicas might still need to queue CCDS messages from the faster replicas.
  - `get_grp_clock_time()`: The thread invokes this member function for each clock-related operation and passes the local clock value that is being proposed for the group clock as an in parameter to this method.
  - `recv_CCDS_msg()`: A received CCDS message is appended to the input buffer that is targeted for this thread.

3.2 The Algorithm

The CCDS algorithm is given in Figures 1 and 2. Each clock-related operation is converted into a CCDS message that is multicast to all of the replicas in a group using a reliable ordered multicast protocol. Each CCDS message contains, in its payload, a group clock value proposed by the sending replica. The clock value contained in the *first* received CCDS message is returned to the application as the consistent group clock value.

Initialization

During initialization, the clock offset at each replica and the round number are set to zero (lines 1-2 in Figure 1), which implies that the CCDS message for the first clock-related operation in each replica contains the physical hardware clock value for that replica.

Consistent Clock Distribution and Synchronization

For each clock-related operation, the physical hardware clock value is retrieved and a local logical clock value is calculated by summing the physical hardware clock value and the clock offset at the replica (lines 3-4 in Figure 1). Then, the consistent clock synchronization handler for the thread is located and the `get_grp_clock_time()` method of the handler is invoked, with the local logical clock value as an in parameter (lines 5-6 in Figure 1). This method invocation blocks until the first matching CCDS message is delivered. Every replica in the group accepts the local logical clock value in that message as the group clock value (as a result of the reliable ordered multicast of CCDS messages). The clock offset is updated by taking the difference of the group clock value and the physical hardware clock value. The group clock value is then returned to the replica (lines 7-8 in Figure 1).

In the `get_grp_clock_time()` method, the round number is incremented each time the method is invoked (line 9 in Figure 1). Any matching CCDS messages for the calling thread that have been received earlier (when the mechanisms could not determine the thread to which those messages should be delivered) are moved from the common input buffer to the local input buffer in the thread (line 10 in Figure 1).

The local input buffer is then checked. If the local input buffer is empty, the mechanisms construct a CCDS message with the appropriate thread identifier, round number and local clock value that is being proposed for the group clock. Then, the mechanisms send the message using the reliable ordered multicast protocol. The calling thread is blocked waiting for the arrival of the first matching CCDS message (lines 11-14 in Figure 1).

When the thread is awakened by the arrival of a CCDS message or if there is a message in the local input buffer, the mechanisms remove the first CCDS message from the local input buffer, extract the local clock value that is being proposed as the consistent group clock value and return the consistent group clock value to the application (lines 15-17 in Figure 1).

On receiving a CCDS message, the mechanisms extract the sending thread identifier and search for the corresponding `CCDS_handler` object. If one is found, the `recv_CCDS_msg()` method of the handler object is invoked and the message is passed to the method as an in parameter. If no handler object is found, the replica has not started the thread yet. In this case, the CCDS message is queued in the common input buffer (lines 1-4 in Figure 2).

```
   On initialization
   begin
1  |  my_clock_offset ←— 0
2  |  my_round_number ←— 0
   end
   For each clock-related operation
   begin
3  |  my_physical_clock_val ←— read from physical hardware clock
4  |  my_local_clock_val ←— my_physical_clock_val + my_clock_offset
5  |  my_CCDS_handler ←— consistent clock synchronization handler
6  |  grp_clock_val ←— my_CCDS_handler.get_grp_clock_time(my_local_clock_val)
7  |  my_clock_offset ←— grp_clock_val − my_physical_clock_val
8  |  return grp_clock_val
   end
   On invocation of the get_grp_clock_time() method
   begin
   |  // The following operation is mutex protected
9  |  my_round_number ←— my_round_number + 1
10 |  move matching CCDS messages from my_common_input_buffer to
   |  my_input_buffer
11 |  if no message in my_input_buffer then
12 |  |  construct a CCDS message with my_local_clock_val,
   |  |  my_round_number, and appropriate thread id
13 |  |  multicast CCDS message
14 |  |  wait until my_input_buffer becomes non-empty
15 |  select the first message in my_input_buffer
16 |  recvd_grp_clock_val ←— consistent clock value in message
17 |  return recvd_grp_clock_val
   end
```

Figure 1: Consistent clock distribution and synchronization. Initialization, sending and retrieving the group clock value.

On invocation of the `recv_CCDS_msg()` method, the CCDS message is checked to see if it is a duplicate (line 5 in Figure 2). If the CCDS message is a duplicate, it is discarded (line 10 in Figure 2). If it is not a duplicate, the message is queued in the handler's local input buffer. If the local input buffer was previously empty, there might be a thread that is blocked waiting for the CCDS message. A signal is sent to wake up a such a blocked thread (lines 6-9 in Figure 2).

Integration of New Clocks

Adding a new replica or restarting a failed replica introduces a new clock. The replication infrastructure ensures that the state transfer, or the synchronization of replica state, occurs when the replicas in the group are in a quiescent state, *i.e.*, are not involved in any processing, including clock-related operations. Therefore, adding a new replica (a new clock) does not interfere with normal consistent clock distribution and synchronization.

It is important to ensure that the newly added replica maintains the property that the group clock increases monotonically. During the recovery process, the new clock must be initialized properly, based on the existing

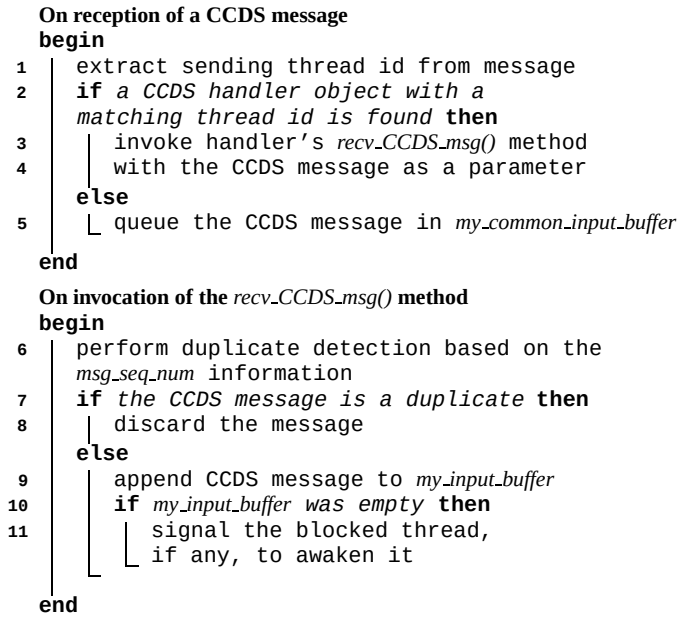

Figure 2: Consistent clock distribution and synchronization: On receiving a CCDS message.

group clock.

When adding a new replica, a synchronization point must be chosen for the state transfer from the existing replicas to the recovering replica [13]. It is generally achieved by a reliable ordered multicast GET_STATE message, which takes a checkpoint.

At the recovering replica, the mechanisms start queuing incoming messages on receiving the GET_ STATE message. At the existing replicas, the GET_STATE message is ordered with respect to the other incoming requests. When the GET_STATE message is scheduled to be delivered to the local replica, a checkpoint of the state of the replica is taken and the checkpoint is transferred to the recovering replica through another reliable ordered multicast message. The checkpoint is applied at the recovering replica once it is received. The replica is then regarded as fully recovered, and the infrastructure begins to deliver the queued messages.

To synchronize the new clock with the group clock, a special CCDS round is executed during the state transfer. In this round, the mechanisms at the existing replicas take a clock value immediately before the checkpoint. The CCDS message sent by the synchronizer of the round is ordered and delivered to all of the replicas in the group, including the recovering replica.

Note that the recovering replica does not participate in the competition for sending the CCDS message. Instead, the mechanisms at the recovering replica perform a clock-related operation as soon as they receive the CCDS message and adjust the offset according to the group clock. At the end of the special CCDS

round, the newly added clock is properly initialized with respect to the group clock.

3.3 Discussion

For passive and semi-active replication, only the primary replica sends CCDS messages. If the primary fails and a backup assumes the role of the primary, that backup might find that a CCDS message has already been received and, consequently, that it does not need to send the CCDS message but, rather, it uses the consistent clock value contained in the CCDS message.

The winner of a consistent clock synchronization round is not necessarily the first replica in the group that conducted the clock-related operation. The order in which messages are multicast depends on the strategy that the group communication protocol uses. In Totem [1], for example, the winner is determined by the relative ordering of the send request and the token visit, together with the position of the replica on the logical ring (as shown in the example in Section 3.4). Nevertheless, a faster replica has a higher probability of becoming the winner of a consistent clock synchronization round. Consequently, the smallest interval is chosen as the interval of the group clock between two subsequent rounds whenever the synchronizer changes, which causes the group clock to drift from real time.

The consistent clock synchronization algorithm presented above applies to a single group of replicas. If there are multiple groups of replicas, the problem of maintaining causal relationships of the group clocks for the different groups arises. This problem is addressed by including the value of the group clock as a timestamp in the user messages multicast to the groups and delaying delivery of a user message at a destination group if the group clock of that group is less than the timestamp carried by the user message. This extension of the algorithm ensures that the causality property is satisfied.

3.4 An Example

An example of the operation of the CCDS algorithm using Totem [1] is shown in Figure 3. The figure shows the progress of real time and three replicas, $replica_1$, $replica_2$ and $replica_3$. Initially, the physical hardware clock at $replica_1$ is synchronized with a real-time clock. The physical hardware clocks at the other replicas are not synchronized. All three replicas start with an offset of 0. In both the figure and the explanation below, we use pc to represent the physical hardware clock, lc to represent the local logical clock, and gc to represent the group clock.

At real time 8:10, $replica_1$ initiates a round of the CCDS algorithm. $Replica_1$ reads its physical hardware clock and sets pc = 8:10. It then adds pc and offset to obtain its local logical consistent clock lc = 8:10, which it multicasts to all of the replicas in a CCDS message. After a short delay, $replica_1$ receives its own CCDS message and determines that gc = 8:10 and then subtracts pc from gc to obtain its offset = 0. $Replica_2$ reads its physical hardware clock and sets pc = 8:15, and then receives the multicast CCDS

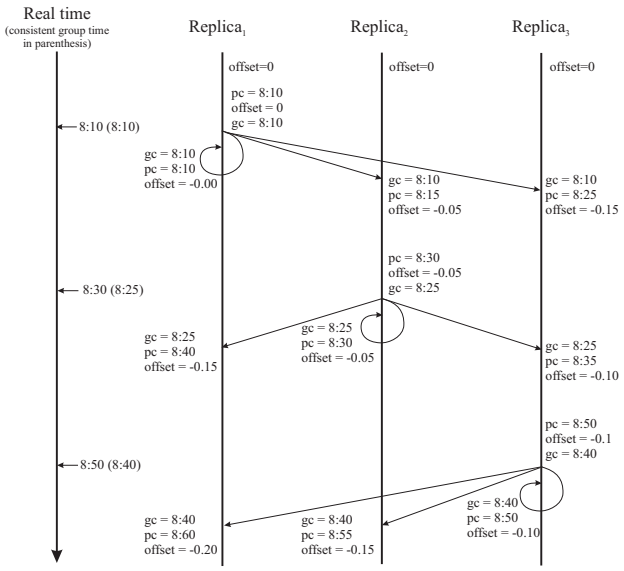

Figure 3: An example of consistent clock synchronization.

message, from which it determines that gc = 8:10. $Replica_2$ then subtracts pc from gc to obtain its offset = -0.05. $Replica_3$ receives the multicast CCDS message, from which it determines that gc = 8:10, and reads its physical hardware clock and sets pc = 8:25. It then subtracts pc from gc to obtain its offset = $gc - pc$ = -0.15.

A short time later, at real time 8:30, $replica_2$ initiates a round of the CCDS algorithm. $Replica_2$ reads its physical clock and sets pc = 8:30, and then adds pc and offset to obtain lc = 8:25, which it multicasts to all of the replicas. After a short delay, $replica_2$ receives its own multicast CCDS message, from which it determines that gc = 8:25. $Replica_2$ then subtracts pc from gc to obtain its offset = -0.05. $Replica_1$ receives the multicast CCDS message, from which it determines that gc = 8:25, and then reads its physical hardware clock and sets pc = 8:40. $Replica_1$ then subtracts pc from gc to obtain its offset = -0.15. $Replica_3$ reads its physical hardware clock and sets pc = 8:35, and receives the multicast CCDS message from which it determines that gc = 8:25. $Replica_3$ then subtracts pc from gc to obtain its offset = -0.1.

At real time 8:50, $replica_3$ initiates a round of the CCDS algorithm. $Replica_3$ reads its physical hardware clock and sets pc = 8:50. It then adds pc and offset to obtain lc = 8:40, which it multicasts to all of the replicas. After a short delay, $replica_3$ receives its own multicast CCDS message, from which it determines that gc = 8:40. $Replica_3$ then subtracts pc from gc to obtain its offset = -0.1. $Replica_1$ reads its physical hardware clock and sets pc = 8:60. It then receives the multicast CCDS message, and determines that gc = 8:40. $Replica_1$ subtracts pc from gc to obtain its offset = -0.2. Similarly, $replica_2$ reads its physical hardware clock and sets pc = 8:55, and receives the multicast CCDS message from which it determines that

gc = 8:40. $Replica_3$ then subtracts pc from gc to obtain its offset = -0.15.

4 Implementation and Performance

4.1 Implementation

The consistent clock synchronization algorithm is implemented in our PluggableFT CORBA infrastructure [21]. The Totem group communication system [1] is used to provide reliable ordered multicasting of the CCDS messages. The implementation uses the C++ programming language. The standard POSIX thread library is used to protect shared data and to block the calling thread for the clock-related operation until the first CCDS message is received for each round.

Most operating systems offer more than one system call to access the physical hardware clock, such as `gettimeofday()`, `time()` and `ftime()`. For each such system call, we assign a unique type identifier so that the CCDS algorithm can recognize and distinguish the system calls. Each CCDS message includes an additional field for this purpose. We use library interpositioning to capture the clock-related operations.

4.2 Experimental Setup

The measurements were performed on a testbed consisting of four Pentium III PCs, each with a 1GHz CPU, 256MBytes of RAM, running the Mandrake Linux 7.2 operating system, over a 100Mbit/sec Ethernet, using e*ORB [20]. During the measurements, there was no other traffic on the network.

Four copies of Totem run on the four PCs, one for each PC. We refer to these PCs as $node_0$, $node_1$, $node_2$ and $node_3$, in the order of the logical token-passing ring imposed by Totem, where $node_0$ is the ring leader. Because one and only one instance of Totem runs on each node, we use the same node designation to refer to the Totem instance on a particular node.

In the experiments, a client makes a remote method invocation on a three-way actively replicated server. The client runs as the ring leader, $node_0$. One replica of the server runs on each of the other three nodes, $node_1$, $node_2$ and $node_3$.

We implemented two different applications in order to study:

1. The overhead of the Consistent Time Service
2. The skew and drift of the Consistent Time Service.

For (1) the client invokes a remote method that returns the current time in two CORBA longs. The server simply calls `gettimeofday()`, which returns the clock value. We measured the probability density functions of the end-to-end latency at the client, with and without the Consistent Time Service running.

Note that replica consistency of the server for this operation cannot be guaranteed if the Consistent Time Service is not activated. Each run contains 10,000 invocations. Note that the client is not replicated and, therefore, the Consistent Time Service is not activated for the client, which makes it possible for the client to obtain the end-to-end latency by reading the local clock value before and after each remote method invocation using the `gettimeofday()` call.

For (2) the remote invocation of the client triggers a sequence of clock-related operations at each server replica. For each run, each server replica performs 10,000 clock-related operations triggered by a remote method invocation of the client.

An empty iteration loop is inserted between two subsequent clock-related operations to simulate a random delay comparable to the token-passing time. Because typical sleep system calls are rounded to an integral number of clock ticks by the operating system, *i.e.,* a multiple of 10ms, they are not used to simulate the random delay.

The number of iterations for the inserted delay is randomly chosen by each replica to be 30,000, 60,000 or 90,000. In our testbed, such numbers of iterations produce a delay ranging from 60 μsec to 400 μsec. The actual delay achieved for the same number of iterations differs slightly for different runs, because of the uncertainty of the CPU scheduling and the CPU time taken by the Totem process. If different replicas choose different numbers of iterations, the difference between the actual delay is larger than one token passing time (the peak probability density of the token passing time on our testbed is approximately 51 μsec [22]) with high probability. We use this arrangement to study the behavior of the Consistent Time Service when the synchronizer rotates randomly among the server replicas.

4.3 Experimental Results

Overhead

The probability density function of the end-to-end latency, with and without the Consistent Time Service, is shown in Figure 4. As can be seen, the Consistent Time Service adds about 300 μsec overhead to the end-to-end latency. This overhead is caused primarily by one additional token circulation around the logical ring. In this extra token circulation, one CCDS message is multicast, ordered and delivered, as is guaranteed by an effective duplicate detection mechanism [22].

For the run shown in Figure 4, which used 10,000 synchronization rounds, the numbers of CCDS messages sent to the network by the three nodes running the server replicas (*i.e.*, $node_1$, $node_2$ and $node_3$) are 1, 9,977 and 22, respectively. For each round, each replica attempts to send one CCDS message. Without duplicate detection and suppression, there would be 10,000 CCDS messages sent by each node, *i.e.,* , a total of 30,000 CCDS messages. With duplicate detection and suppression, only one CCDS message is sent per

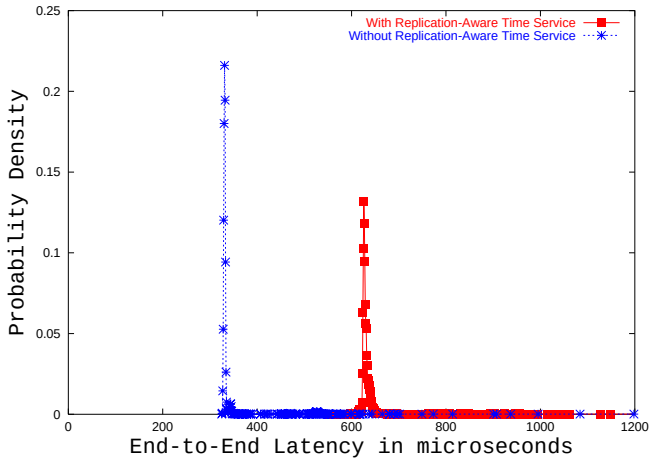

Figure 4: The probability density functions of the end-to-end latency measured at the client, with and without the Consistent Time Service.

synchronization round and, thus, the total number of CCDS messages sent is 10,000, the same as the number of synchronization rounds.

The overhead appears to be high because the server performs no significant processing in our experiment. If each remote invocation on the server results in 2 milliseconds of processing (on the application side), the cost of the Consistent Time Service of 300 microsecond would be only 15%.

Measurements of Skew and Drift

The time intervals of the first 20 rounds of the clock-related operations are shown in Figure 5 (a). As can be seen from Figure 5(a), the synchronizer of the consistent clock synchronization algorithm is constantly changing from one replica to another.

The two major factors in each time interval are: (1) the inserted delay, and (2) the time needed to perform the consistent clock distribution and synchronization. The larger than usual time interval between the 19th and 20th rounds is caused by (2) rather than (1). For (2), there are data points collected with long latency, albeit with very low probability. This larger than usual overhead in consistent clock distribution and synchronization is due to Totem [5, 17].

The changing of the clock offset at the replica that wins the first round (*i.e.*, $replica_2$ running on $node_2$), as multiple rounds proceed, is shown in Figure 5 (b). As expected, the clock offset occasionally (at rounds 2, 5 and 15) increases when the time interval of the last two clock-related operations at the current round synchronizer is larger than that at $replica_2$. Because this increase happens quite rarely in our experiments and as predicted by the consistent clock synchronization algorithm, the overall trend for the offset is seen to

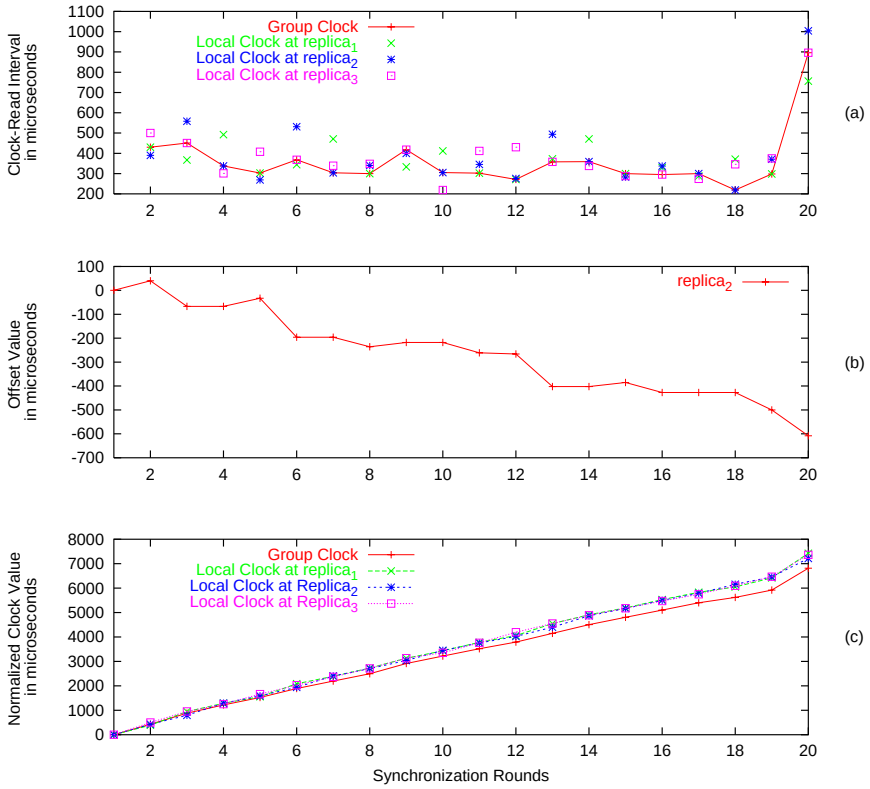

Figure 5: The first 20 rounds of the consistent clock synchronization algorithm: (a) Interval between two clock-related operations at each replica using the physical hardware clock and the group clock. (b) Clock offset of the first round winner. (c) Normalized physical hardware clocks at each replica and the group clock.

be decreasing. For the same reason, the group clock runs slower than real time, as can be seen in Figure 5 (c).

For the purpose of investigating the drift of the group clock, the group clock is compared with the physical hardware clock at each $replica_i$ and the physical hardware clock values are normalized by subtracting the value obtained for the physical hardware clock in the initial round. The normalized physical hardware clock values and the group clock are shown in Figure 5 (c). The normalized physical hardware clocks at the three replicas cannot be distinguished in the figure. The difference in the values read from the physical hardware clocks for the same round is on the order of one or two hundred microseconds (as can be seen in Figure 5 (a)), which is significantly smaller than the time-scale used in the plot for Figure 5 (c).

5 Conclusion and Future Work

We have presented the design and implementation of a Consistent Time Service for fault-tolerant distributed systems that renders deterministic readings of the physical hardware clocks of the replicas within a group.

The Consistent Time Service, described in this paper, is important not only for active replication during normal operation, but also for passive replication and semi-active replication, when the primary fails and a backup replica takes over, to ensure a consistent monotonically increasing clock.

The Consistent Time Service applies to both a single group of replicas and multiple groups of replicas. The problem of maintaining causal relationships of the consistent group clocks for the different groups is addressed by including the value of the consistent group clock as a timestamp in the user messages multicast to the different groups and delaying delivery of a user message at a destination group if the group clock of that group is less than the timestamp carried by the user message.

There are many interesting open issues related to clock synchronization in the context of a Consistent Time Service, including how to synchronize the group clock with an external real time source, how to synchronize the clocks of replicas within a group, and how to correct the systematic drift of the group clock caused by synchronizer changes in different rounds. The investigation of each of these issues constitutes a research topic in its own right.

References

[1] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, November 1995.

[2] C. Basile, K. Whisnant, Z. Kalbarczyk, and R. Iyer. Loose synchronization of multithreaded replicas. In *Proceedings of the IEEE International Symposium on Reliable Distributed Systems*, pages 250–255, Suita, Japan, October 2002.

[3] K. P. Birman and R. van Rennesse. *Reliable Distributed Computing Using the Isis Toolkit*. IEEE Computer Society Press, 1994.

[4] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, February 1996.

[5] R. K. Budhia, L. E. Moser, and P. M. Melliar-Smith. Performance engineering of the Totem group communication system. *Distributed Systems Engineering*, 5(2):78–87, June 1998.

[6] Intel Corporation. *Desktop performance and optimization for Intel Pentium 4 processor*, February 2001.

[7] R. Jimenez-Peris and S. Arevalo. Deterministic scheduling for transactional multithreaded replicas. In *Proceedings of the IEEE 19th Symposium on Reliable Distributed Systems*, pages 164–173, Nurnberg, Germany, October 2000.

[8] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, pages 25–40, February 1989.

[9] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, 1985.

[10] D. L. Mills. Internet time synchronization: The Network Time Protocol. *IEEE Transactions on Communications*, COM-39(10):1482–1493, October 1991.

[11] S. Mullender, editor. *Distributed Systems*. ACM Press, second edition, 1993.

[12] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded CORBA applications. In *Proceedings of the IEEE 18th Symposium on Reliable Distributed Systems*, pages 263–273, Lausanne, Switzerland, October 1999.

[13] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Strongly consistent replication and recovery of fault-tolerant corba applications. *Computer System Science and Engineering Journal*, 17(2):103–114, March 2002.

[14] D. Powell, editor. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, 1991.

[15] L. Rodrigues, P. Veríssimo, and A. Casimiro. Using atomic broadcast to implement a posteriori agreement for clock synchronization. In *Proceedings of the IEEE 12th Symposium on Reliable Distributed Systems*, pages 115–124, Princeton, NJ, October 1993.

[16] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, 1987.

[17] E. Thomopoulos, L. E. Moser, and P. M. Melliar-Smith. Latency analysis of the Totem single-ring protocol. *ACM/IEEE Transactions on Networking*, 9(5):669–680, October 2001.

[18] P. Verissimo. Ordering and timeliness requirements of dependable real-time programs. *Journal of Real-Time Systems*, 7(2):105–128, 1994.

[19] P. Veríssimo and L. Rodrigues. A posteriori agreement for fault-tolerant clock synchronization on broadcast networks. In *Proceedings of the IEEE 22nd International Symposium on Fault-Tolerant Computing*, pages 527–536, Boston, MA, July 1992.

[20] Vertel Corporation. *e*ORB C++ User Guide*, December 2000.

[21] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. Design and implementation of a pluggable fault tolerant CORBA infrastructure. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, Special Issue on Dependable Distributed Systems, 7(4):317–330, October 2004.

[22] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. End-to-end latency of a fault-tolerant CORBA infrastructure. In *Proceedings of the IEEE International Symposium on Object-Oriented and Real-time Distributed Computing*, pages 189–198, Washington, DC, April 2002.