Latecomer and Crash Recovery Support in Fault Tolerant Groupware

Mihail Ionescu and Ivan Marsic
Center for Advanced Information Processing (CAIP), Rutgers University
96 Frelinghuysen Road
Piscataway, NJ 08854-8088, USA
+1 732 445 0549
{mihaii, marsic}@caip.rutgers.edu

Abstract
Distributed collaboration systems must allow dynamic joining and leaving of sessions and therefore must support latecomers and crash recovery. We present two distributed algorithms for supporting latecomers and crash recovery and evaluate them within the DISCIPLE framework for collaboration. Both algorithms are generic, independent of the application semantics, and can work with arbitrary JavaBeans that are shared in DISCIPLE. The second algorithm requires a minimum assistance from the application by implementing a framework-specified interface. The main advantages of these algorithms are their transparency to the user and their fault tolerant nature. Our algorithms are entirely distributed and resilient to site failures in the sense that the remaining sites can continue without interruption. The framework chooses dynamically at run time which algorithm to use depending on whether or not the application implements the interface. Two example applications, one suitable for each algorithm, are also presented.

Keywords: Groupware, fault-tolerant computing, shared electronic workspaces.

Name of Magazine: IEEE DS Online.

Note: Proof of Correctness in Section 2.1 and Section 4 could be boxed as “sidebars.”
Latecomer and Crash Recovery Support in Fault Tolerant Groupware

Mihail Ionescu and Ivan Marsic

Real-time collaborative systems are distributed systems where the actions of one user must be instantaneously propagated to all the users participating in a session. The applications include shared electronic whiteboards, chat rooms, audio/video conferencing, collaborative design, collaborative virtual worlds, and multiplayer games.

The composition of the participants in a collaborative session can change dynamically. A session may start with some users and other users may join the session later while some may leave and join again. Unlike stateless collaborative applications (videoconferencing, discussion groups), in state-full collaborative applications (whiteboard, text editor, etc.) it is mandatory that the system offers latecomer and crash recovery support.

The researchers take one of the two main directions: logging and playback of events or exporting the application state to the newcomer. Both have advantages and shortcomings. Event logging and playback imposes minimum requirements on the applications and allows the latecomers to see how the existing participants reached the present state. However, it may require excessive amount of storage since sessions may last for a long time. Bringing a latecomer up to date may also take long time, for the same reason. The other technique, exporting the application state to the latecomer, has minimum storage requirements and takes relatively short time. However, the latecomer cannot see how the present situation developed. Also, the application must provide for exporting its data structures.

Latecomer and crash recovery support should satisfy most or all of the following, sometimes contradictory, requirements:

- resilience to individual site failures
- transparency to the user and application developer, being a general-purpose part of a collaboration infrastructure
- high interactivity with short latecomer updating time
- economical resource usage, particularly of storage
- flexibility in history review—from the final state only to the entire history.

The system presented here meets the requirements by implementing two alternative algorithms for latecomer support. The main characteristics of our solution are:

1. Fault-tolerance due to distribution of the latecomer support across multiple hosts.
2. Guaranteeing the correct updating of the newcomer in a finite time, with minimal system constraints.
4. A simple method to adjust the number of sites involved in latecomer support, so to maintain the right balance between performance and cost in terms of resources.

The solution is implemented and evaluated in our framework for synchronous collaboration, called DISCIPLE. DISCIPLE supports building and sharing collaborative applications from single-user beans (JavaBeans-compliant components, applets, and applications). DISCIPLE can also run as a Java applet in a Web browser.

1 Collaboration Framework

A collaborative configuration is defined as a structure that holds the active sites, \( \Gamma = \langle S_{a1}, \ldots S_{aw} \rangle \). Each site hosts an application that communicates with the applications from remote sites. We can define the application state in two different ways. First, we can define it as a pair \( S = \langle id, O \rangle \), where \( id \) is a unique identifier associated with the site (e.g., its IP address) and \( O \) is an ordered set of instances of operations \( Op_1, Op_2, \ldots Op_m \) along with the input parameters. Second, we can define the state as a structure \( S = \langle id, o \rangle \), where \( id \) is the unique identifier and \( o \) is an array that represents the serialized application state.

The configuration of sites is correct or consistent if for any two sites \( \alpha, \beta \in \Gamma \) the following holds: If at any time the system were to become quiescent with no
messages in transit between the sites, then their application states must be the same, $S_a = S_b$. A key issue in distributed frameworks is maintaining the correctness of the configuration in the presence of concurrent access.

1.1 Total Ordering of Events

We consider that the events are transmitted reliably but no order is guaranteed at receiving. Since the operations defined by applications generally do not commute, the events received out of order may adversely affect the configuration correctness.

Given the events $e_\alpha$ and $e_\beta$, generated at the sites $\alpha$ and $\beta$, then $e_\alpha$ precedes $e_\beta$ if and only if: (i) $\alpha = \beta$ and $e_\alpha$ was generated before $e_\beta$ or (ii) $\alpha \neq \beta$ and the execution of the $e_\alpha$’s operation at $\beta$ happened before the generation of $e_\beta$. We define the precedence property: if an event $e_\alpha$ precedes an event $e_\beta$, then the execution of the $e_\alpha$’s operation happens before that of $e_\beta$ at every site.

The precedence property does not guarantee correctness of the configuration. One possible solution adopted in DISCIPLE is to extend the partial ordering offered by the precedence property to a total ordering. Given a configuration $\Gamma$ with each site $\alpha$ sending its own events $e_{\alpha 1}, e_{\alpha 2}, \ldots e_{\alpha k}$, the total ordering is present if every site receives all the events in the same order. We say that times $t_\alpha$ and $t_\beta$ at the sites $\alpha$ and $\beta$ are logically equivalent if $\alpha$ receives an event $e$ at local time $t_\alpha$ and $\beta$ receives the same event at local time $t_\beta$ assuming the total ordering of the events.

Each event is labeled so to distinguish the collaborative events from those generated during the latecomer updating. The label $\ell$ is assigned as follows: the value 1 is associated with events that are sent due to the user actions, the value 2 is associated with the membership events and the value 3 with the events used for latecomer and crash recovery support.

1.2 Internet Implementation

True multicast is not possible in today’s Internet since many network routers and firewalls do not support it. To overcome this problem, we implemented a simulated multicast protocol based on TCP using an asymmetric total ordering algorithm. The first site to start in a configuration becomes a centralizer for the configuration. Each new site sends an authentication message to the centralizer upon joining the session. The centralizer accepts the new site and establishes a connection with it. When a site wants to send an event, it sends the event to the centralizer, which broadcasts it to all the sites. The asymmetric total ordering is simpler to implement compared to the symmetric one, but it is not very scalable. However, collaborative editing sessions typically do not involve more than 5-10 persons, especially for the applications presented in this paper.

2 Late-comer and Crash Recovery Support

The algorithms presented below are based on the two general approaches for latecomer support.

2.1 Event Logging and Replay

This algorithm essentially works as follows. Each site keeps a log of all the received events that change its application state. In this case the application state is defined as $S = <id, O>$. When a newcomer joins the session, the newcomer multicasts a special event (denoted by join_query) announcing its presence and requesting the shared application state. All of the active sites that receive this event reply with a confirmation message. The latecomer ignores all but the first message and establishes a point-to-point connection with its sender, say the site $\beta$, and receives all the events that were logged in the $\beta$’s log. After $\beta$ sends all the events in its log, it sends a termination message to the latecomer.

Data Structures

Events. Events are tuples of the form $e = <\alpha, d, \ell>$, where $\alpha$ is the sending site’s identifier, $d$ is the information associated with the event (operation and parameters) and $\ell$ is the event’s label. The event $e$ has three methods for getting the id, the data, and the label: $id(e), data(e), label(e)$.

Event log. Each site $\alpha$ maintains an event log $L_\alpha$ of the configuration events. The log is ordered so it is possible to step through it in the insertion order. Recorded along with the event is the local time when it was received.

Late event log. Each site $\alpha$ maintains an additional event log, $LL_\alpha$ which is also ordered. This log is used for backing up the events received at the site while it updates a latecomer.

* Note that the point-to-point connection does not participate in the total ordering process.
Framework state. Each site $\alpha$ maintains a state $S_\alpha$ (different from the application state). A site can be in one of the following states: late, late_wait, (being) updated, normal, normal_wait or updating, as illustrated in Figure 1. A site is in the state late at the startup and transitions to normal after all of the events logged at an active site are received and executed. In the normal state, the user interface is enabled for interaction. The states updating and normal_wait are visited when the site sends the events from its private log to a latecomer.

Query Number and Map. Each site $\alpha$ maintains a query sequence number $q_\alpha$ used to prevent the “starvation” of the latecomer. Assume a configuration with three sites $\alpha$, $\beta$, and $\gamma$. Assume a new site $\delta$ wants to join and $\alpha$ responds the first to the join_query event and initiates point-to-point communication with $\delta$. Suppose that immediately upon initiating the connection, $\alpha$ crashes obeying a fail-stop model\footnote{A fail-stop process is one that, when it is about to fail, changes to a state that permits other processes to detect that a failure has occurred and then halts. It never produces value, omission or timing failures.}. After the timeout expires, $\delta$ resends the join_query event. Suppose that in the meantime $\alpha$ recovered (by having a high speed connection with $\beta$, for example) and again $\alpha$ responds the first to $\delta$ but crashes after a short time. If this persists the site $\delta$ will starve and will never be able to reach the shared state even if it could do so by establishing a connection with $\gamma$, for example. We solve this by adding $q_\alpha$ to each join_query event. A site answers to the join_query event only if it has received all the preceding join_query events. Each site maintains a map $M_\alpha$ containing the query numbers of the current latecomers. The query number can be obtained from a join_query event $e$ using the method $\text{qn}(e)$. We say that a site that responds to join_query event sent by the site $\beta$ is a responding site with respect to $\beta$.

It may happen that none of the remaining configuration members is able to respond to join_query because they did not receive all the preceding messages. Consider an example with two sites $\alpha$ and $\beta$ and a site $\gamma$ trying to join. Assume that $\alpha$ responds to the first join_query and fails during the updating. The site $\gamma$ resends join_query and $\beta$ responds but also fails during the updating. When the next join_query event is sent by $\gamma$ no site will be able to respond even if $\alpha$ and $\beta$ recovered and are active members. The site $\gamma$ will assume that it is the first one in the configuration which is not true. Therefore, when a latecomer does not receive a response_backup event after one or more unsuccessful attempts, it resets its query number and starts again by sending a join_query event.

Algorithm

The inset below gives a more precise specification of the algorithm executed at an arbitrary site $\alpha$.

```plaintext
L_\alpha ← empty
S_\alpha ← late
q_\alpha ← 0
M_\alpha ← empty
while (still active) do
  switch (S_\alpha)
    case late:
      join:
        create a join_query event with
        the current $q_\alpha$ and broadcast it
      S_\alpha ← late_wait
    case late_wait:
      wait for response $r$
      switch ($r$)
        case response_backup:
          $\delta ← \text{id}(r)$
          send a connection event to site $\delta$
          $S_\alpha ←$ updated
        case timeout:
          if ($q_\alpha > 0$)
            $q_\alpha ← 0$
            $S_\alpha ←$ late
            goto join
          else
            $S_\alpha ←$ normal
    case updated:   // being updated
      do forever
        wait for event $m$ from site $\delta$
        if (timeout)
          $L_\alpha ←$ empty
          $S_\alpha ←$ late
          goto join
```
\[ q_\alpha \leftarrow q_\alpha + 1 \]

if \((m \text{ is end_backup})\)
  break;
\[ \text{data} \leftarrow \text{data}(m) \]
\[ \ell \leftarrow \text{label}(m) \]
\[ L_\alpha \leftarrow L_\alpha + \langle \delta, \text{data}, \ell \rangle \]
execute the \(m\)'s operation
\] od
\[ S_\alpha \leftarrow \text{normal} \]

2 case normal:
  while \((S_\alpha = \text{normal})\) do
    receive event \(m\)
    \[ \ell \leftarrow \text{label}(m) \]
    \[ \delta \leftarrow \text{id}(m) \]
    switch \((\ell)\) // label value
    case 1:
      \[ L_\alpha \leftarrow L_\alpha + m \]
      if \((\delta = \alpha)\) // local event
        broadcast \(m\) to all other sites
      for each \(e\) in \(LL_\alpha\)
        execute the \(e\)'s operation
      end for
    case 3:
      if \((m = \text{join_query})\)
        \[ q \leftarrow qn(m) \]
        find \(q_\beta\) in \(M_\alpha\) for site \(\beta\)
        if \((q \leq q_\beta + 1)\)
          send \text{response_backup} to \(\delta\)
          increment \(q_\beta\)
          \[ S_\alpha \leftarrow \text{normal_wait} \]
        fi
      fi
    end switch
  od
3 case normal_wait:
  receive event \(m\) from site \(\beta\)
  switch \((m)\)
  case (connection): \[ S_\alpha \leftarrow \text{updating} \]
  case (timeout): \[ S_\alpha \leftarrow \text{normal} \]
  end switch
4 case updating:
  \[ e = \text{getNext}(L_\alpha) \]
  while \((e \neq \text{nill})\) do
    send \(e\) to site \(\delta\)
    receive event \(m\);
    if \(!(\ell = \text{timeout})\)
      if \((\text{label}(m) = 1)\)
        \[ L_\alpha \leftarrow L_\alpha + m \]
        \[ LL_\alpha \leftarrow LL_\alpha + m \]
      \(e = \text{getNext}(L_\alpha)\)
    od
    send \text{end_backup} to site \(\delta\)
  \[ S_\alpha \leftarrow \text{normal} \]
5 end
6 od

Analysis

The algorithm is completely distributed and resilient to site failures. However, each site must keep an event log in the memory (or on the disk). This may be undesirable since some hosts may have lower capabilities. We solve this problem by allowing the user at runtime to decide whether his or her site should keep the log or not. If the site does not maintain the event log, it lacks \text{updating} state and ignores the \text{join_query} events. In this case only a subset of sites are \textit{involved} in the latecomer support (Figure 2).

On one end we can have all the sites involved and on the other end the involved set comprises only one site. We found the latter, “centralized,” architecture suitable for an environment where there exists a high-end server (called \textit{central site}), and the rest are low-end computers (e.g., laptops). However, the crash of the central site completely eliminates the latecomer and crash recovery support. To solve this, we run a distributed algorithm that checks whether the central site is alive and if not, activates a backup server in its stead. The backup server also keeps the event log, but does not respond to the \text{join_query} events before activated. It listens to the multicast channel and logs all the events. This technique resembles passive replication\textsuperscript{6} with the difference that in our case the backup server actively listens for events.

The sites communicate with each other using a heartbeat protocol. Each site periodically sends a message informing the others that it is still alive (membership events labeled with 2). Every site can at a given quantum of time determine which sites left or joined the session. Also, each site maintains the \text{serverid} as the id of the central site.

When a site discovers that the central site left the session, it broadcasts a special event, \text{query_server}.

![Figure 2. A set of active sites that are involved in the latecomer and crash recovery support. The sites marked “weak” do not participate in this process.](image-url)
After the backup server receives this event from all the remaining sites, it broadcasts a server_set event. Every site that receives this event updates its serverid. The backup server now becomes the central site and starts responding to the join_query events. All the user actions are blocked between the times when a site discovers that central site left the session and until the serverid event is received.

In an asynchronous environment like Internet it is not possible to differentiate between slow and failed members. Our failure detection mechanism is based on the assumption that, if a site does not send the heartbeat for a certain number of periods, it means that the site crashed. The same problem appears when a new site joins the configuration and does not receive any response_backup event. In our implementation, the site simply assumes itself being the first one. However, there might exist other members which could not answer because of the network partitions. Supporting partitioned networks and merging the corresponding states is a direction of our continuing work.

If the set of the involved sites is small, the updating process may take longer if many latecomers try to join simultaneously. This tradeoff between performance and cost is characteristic for distributed systems. Our framework offers the flexibility to dynamically increase the number of involved sites, so the right balance can be achieved.

We assume that the events occurring after the last backup event sent by the updating site are received by the latecomer without loss, so that the latecomer can execute those events in the same order as the other sites. However, this assumption may not always be correct. The latecomer may receive some of the new events before the end_backup event while the updating site receives it after it sent the end_backup event (Figure 3). This is because the events sent on the point-to-point connection are not totally ordered. According to the algorithm, these events would be ignored. One way to solve this problem is to have the latecomer and the updating site record the logically equivalent times. (The latecomer too receives its own join_query event since the event is sent to the multicast channel which is totally ordered.)

**Proof of Correctness**

Here we consider the case where a latecomer joins a correct configuration. The key assumption is that there are no messages in transit when the latecomer joins. The algorithm works for the general case, but the proof is more complex.

For the sake of simplicity we make the following assumptions about the communication medium:

1. If site $\alpha$ sends an event $e_{a1}$ (using the multicast channel or a point-to-point connection) and after a short period of time it sends an event $e_{a2}$, other sites receive the events in the order $e_{a1}, e_{a2}$.
2. The possible failure of updating sites obeys the fail-stop failure model, so the failure can be determined using a timeout mechanism.

Consider having a correct configuration $\Gamma = <\alpha_1, \ldots, \alpha_n>$ at time $t_0$ and a latecomer, site $\beta$, wants to join.

**Lemma 1.** Consider a subset $A \subseteq \Gamma$ at time $t_0$, where $|A|=C$, $A=\{\alpha \in \Gamma \mid \alpha$ is involved and $\alpha$ is responding with respect to $\beta\}$. Obviously, $C \leq n$, and we make the assumption that $C \geq 1$ at any time. Suppose also that at time $t_0$ each site $\alpha \in \Gamma$ executes the same sequence of events (since $\Gamma$ is a correct configuration at that time), denoted by $E_0=\{e_1, e_2, \ldots, e_p\}$, where $p$ is the number of events generated by the active sites until $t_0$. Then, for every site $\tau \in A$, $L_{t_0} = E_0$. 

![Figure 3. Example for ignoring the new event by the latecomer.](image-url)
Proof. The proof follows from the definition of configuration correctness. Consider an arbitrary site \( \tau \in A \). Since \( A \subseteq \Gamma \), this implies \( \tau \in \Gamma \), so all the events from \( E_0 \) were executed at \( \tau \). But, according to the algorithm, each executed event is also inserted in the log \( L_\tau \) for any involved site. So \( L_\tau = E_0 \), for every \( \tau \in \Gamma \).

Lemma 2. We assume the notations as in Lemma 1. If a latecomer \( \beta \) joins \( \Gamma \) at time \( t_0 \), after a finite time all the events from \( E_\beta \) will be executed in the order \( e_1, e_2, \ldots, e_p \) at \( \beta \).

Proof. Since \( C \geq 1 \) at any time, we can assume that \( \beta \) starts a point-to-point communication with a site \( \tau \), \( \tau \in A \). First, consider that \( \tau \) does not fail until it sends all the events from \( L_\tau \) to \( \beta \). Following the first assumption, the events are received by \( \beta \) in the same order as sent by \( \tau \). Because we assumed that \( \tau \) does not fail until it sends all the events from \( L_\tau \), after a finite period of time \( \beta \) receives and executes all the events from \( L_\tau \). But, according to Lemma 1, \( L_\tau = E_0 \), so the lemma is proved.

Assume now that \( \tau \) fails before sending all the events from \( L_\tau \). According to the assumptions, the failure obeys the fail-stop model, so the timeout mechanism detects the failure, and the join_query event is sent again. Since we assumed that \( C \geq 1 \) at any time, there exists another involved site that responds with a response_backup event. The key observation here is that \( \tau \) will not respond anymore to join_query. Using the induction on \( C \) and the fact that \( C \geq 1 \) at any time, the lemma is proved.

Theorem. Assume that the configuration \( \Gamma \) is correct at time \( t_1 \). The algorithm is correct in the sense that, at \( t_1 \) the latecomer has the same application state as the prior sites.

Proof. According to Lemma 2, after a finite period of time the latecomer receives all the events from \( E_0 \). We can define the set \( E_1 = \{e_1, e_2, \ldots, e_p\} \) in the same way as we defined the set \( E_0 \), where \( q \geq p \). Let us denote by \( E = E_1 \setminus E_0 \). The events from \( E \) received before the latecomer has received all the events from \( E_0 \) are buffered by the latecomer. \( \Gamma \) is correct at time \( t_1 \) so no messages are in transit. This implies that all the events from \( E_1 \) are received by the latecomer at time \( t_1 \). This proves the theorem.

Crash Recovery

Crash recovery is based on the latecomer support. In addition, an increasing sequence number is attached to each received event, and each site keeps the sequence number of the last processed event. Before DISCIPLE delivers an event to the application, it first backs it up on the stable storage along with the current sequence number. As mentioned above, we assume that sites crash obeying the fail-stop model and the network is reliable (it will not partition). When a site wants to rejoin the configuration, it loads the events first from the local storage and sends the restored sequence number to the updating site, which responds with only the events with sequence numbers greater than that one.

Limitations

The algorithm is generic and requires no support from the application. It also allows the latecomer to replay all the events, so the latecomer can see the history of the collaborative work. However, the algorithm features a continuously growing data structure (the event log). The algorithm may be very inefficient. Suppose that the sites work on a shared whiteboard creating many figures and modifying their attributes, but in the end the participants realize that they did everything wrong, so they erase the whiteboard. If a latecomer joins the session at this moment, it will execute many events only to end up with a clear whiteboard. Additionally, if the number of logged events is large, the updating site will be in the updating state for a significant amount of time. The user at the updating site will either not be able to make any actions during this period or, if multithreading is used, the response may be slowed down.

The second algorithm remedies these problems.

2.2 Exporting Application State

In this algorithm, the application state at a site \( \alpha \) is defined as \( S = \langle \alpha, o \rangle \), where \( o \) is an array of bits representing the serialized application state. The main idea of this algorithm is to specify an interface to be implemented by the applications wishing to have latecomer support. When a site decides to send the current shared application state to a latecomer, it calls the interface method on the local application to retrieve its current state as an array of bits. The array is encapsulated in a predefined type of event and sent to the latecomer.

The basic structure of this algorithm is very similar to the previous one, so we outline only the main
differences. The blocks between the numbered lines are changed accordingly.

1 wait for event serialize from site δ
   if (timeout) goto join
   execute the event’s operation
2 $S_δ \leftarrow$ normal
3 case normal:
   while ($S_δ ==$ normal) do
      receive event $m$
      $\ell \leftarrow$ label($m$)
      $\delta \leftarrow$ id($m$)
      switch ($\ell$)
      case 1:
         if ($\delta = \alpha$)
            broadcast $m$ to all other sites
            for every $e$ in $LL_\alpha$
               execute the $e$’s operation
         end for
      execute the $m$’s operation
4 case updating:
   do in parallel:
      obtain serialized application state
      | |
      receive event $m$
      if (!timeout)
         if ($\text{label}(m) == 1$)
            $LL_\alpha \leftarrow LL_\alpha + m$
         od
      send serialize event to site $\delta$
      $e = \text{getNext}(LL_\alpha)$
      while ($e \neq \text{nill}$) do
         send $e$ to site $\delta$
      receive event $m$
      if (!timeout)
         if ($\text{label}(m) == 1$)
            $LL_\alpha \leftarrow LL_\alpha + m$
         $e = \text{getNext}(LL_\alpha)$
         od
6 send end backup to site $\delta$

Serializing the application state can take a significant amount of time for a complex application state. During this time it is possible that other sites generate events, so we still maintain the $LL$ log. One message is sufficient to update the latecomer’s state. Because message sending is asynchronous, the probability that new events will arrive is small, so the site will be in the updating state for a shorter amount of time, reducing the period when the local application is frozen or slowed down.

The main advantage of this algorithm is that the event log $L$ is not maintained. Many times there is no reason to maintain the log of events, as the latecomer may not be interested in the session history. However, the problem will arise if the latecomer is interested in the session history. Also, the generality of the previous algorithm is lost in the sense that the application must be aware of the latecomer support (it has to implement the framework-specific interface).

In this algorithm, crash recovery is completely provided by the latecomer support. When a site resumes a session, its old application state is entirely replaced by the new one obtained from an active site.

3 Implementation and Performance

We evaluated the performances of the framework both in a LAN environment and in the global Internet. Two collaborative applications were used:

Medical Imaging application was developed for interactive indexing and retrieval of pathology images and used with a large image database and a real-time microscope. It performs color, shape and texture analysis of blood-cell images and detects and identifies the cells afflicted by leukemia by comparing them to a database of diagnosed cases.

Whiteboard is a classic shared whiteboard, in which multiple users collaborate to create a complex 2D drawing.

In the Medical Imaging application, the session history is important for the latecomer’s understanding of the current session state, as shown in Figure 4. Replaying all the events seems to be the only feasible solution. Thus the application does not implement the latecomer support interface of the second algorithm.

In the Whiteboard case, either of the described algorithms can be used. A latecomer may be interested in replaying all the events or going directly to the final

Figure 4. The Medical Imaging application and some of the image analysis steps seen by a latecomer.
shared whiteboard state. We choose at runtime what algorithm to use as follows. As mentioned above, DISCIPLE determines if the application implements the latecomer interface. If so, DISCIPLE offers the user the choice to use either of the algorithms for supporting the latecomers.

Let us assume that during a collaborative session the shared application state becomes as in the upper row in Figure 5. If the second algorithm is chosen, only the final application state will be displayed when a latecomer joins the session. If the first algorithm is used, the sequence of all events will be replayed.

### 3.1 Performance in a LAN Environment

We consider the Whiteboard as the base application used to compare the overall performance of the two algorithms. For simplicity, we assume that the Whiteboard can handle a single type of objects (say the Rectangle) and defines four operations on the object: Create, Transform (translate, rotate, scale), ModifyAttribute (color, line width, etc.), and Delete.

The size of the serialized application state depends on the difference between the number of Create and Delete operations. A test scenario contains a fixed number of operations $N_{\text{op}}$ (100 in this case), of which the number of Create operations varies from 1 to $N_{\text{op}}$. If $N_{\text{Create}} < N_{\text{op}}$, the remaining operations are randomly distributed between Transform, ModifyAttribute, and Delete. We define the *update time* as the time required to bring the latecomer up to date after it issues a `join_query` event. The events are played immediately one after another rather than trying to simulate the real time as in systems which additionally record speech conversation. The average results for ten executions of the test scenario are given in Figure 6.

We implemented two versions of the event-logging algorithm. In the first one, the events are logged onto the disk and in the second one they are kept in the memory. As expected, the performance of this algorithm does not change with the relative increase of the Create operations. However, the update time for the state exporting algorithm increases almost linearly with the relative increase of the Create operations. We notice that the second algorithm has almost the same performance as the first one (with the events logged in memory) when all the operations are Create.

### 3.2 Performance on the Internet

We implemented the Medical Imaging application as a Java applet, as shown in Figure 7. In this configuration, the centralizer and the Web server run on the same host.

We are mainly interested in the scalability of this centralized scheme. We varied the number of latecomers that concurrently want to join the collaborative session and measured the time required for updating their state. The test configuration was composed of up to seven computers on a 10 Mbps Ethernet LAN. The average results are given in Figure 8.

We notice that the performance of the system scales well
when the number of concurrent joins is relatively small. However, as the number of concurrent accesses increases, the update time increases faster than linearly (starting with 4 latecomers). This is because of hardware limitations of the Web server, since each connection requires a separate thread.

4 Related Work

Fault tolerance and failure recovery research have long history in distributed systems community. These issues received the most attention in the distributed databases field. Fault-tolerance methods using message logging and checkpointing\(^1\) are particularly interesting since they are general-purpose and do not require excessive amounts of storage. Examples of such techniques are dependency tracking\(^11\) in the process of rollback to a common state or using transitive and indirect dependencies\(^12\) to obtain the most recent consistent global state after recovery. Virtual synchrony\(^8,13\) provides a formal definition of group communication in presence of failures by offering a well-defined guarantee regarding the order of messages and of failure notifications. There are other systems\(^14,15\) that handle the member failure during state update.

Crash recovery schemes for general distributed systems represent a basis for groupware latecomer and crash recovery schemes. The event-logging scheme derives from the message-logging scheme. However, straightforward application is not possible due to the particularities of groupware systems. In groupware, the users are mainly interested in the responsiveness of the system and maintaining the correctness of the configuration. The rollback algorithms are of no use because of the confusion they introduce to the users. Also, the events transmitted in the network are usually small so the total size of the event log may not be excessive.

The architecture of the groupware system plays an important role in the design of latecomer support. Most of the systems can be classified into centralized\(^16,17,18\) vs. replicated architectures\(^19,20,10\). Supporting a latecomer in a centralized architecture is easier and usually means exporting the model’s application state to the latecomer’s view for display (e.g., in Suite\(^17\)). Most of the solutions take either event logging and replaying approach\(^16,10,18\) or exporting application state\(^17,19,20\) but not both or their combination. In some cases\(^19,20,18\) the latecomer support module is centralized, although the overall system architecture is replicated. In GroupKit\(^19\) the latecomer service is provided by a dedicated server. As with any centralized system, a failure of the server affects the whole configuration. Corona\(^18\) offers fault tolerance by maintaining logs on the stable storage on the server, as well as on the clients (in case the server crashes). Most of these systems do not tolerate failures during latecomer updating.

A system by Manohar and Prakash\(^10\), which is based on event logging, provides a VCR-like interface, so that the late user can pause, skip and fast-forward the replay.

---

\(^1\) Checkpointing is a technique in which during the normal execution application states are periodically saved on a stable storage and used during the recovery for rollback to an earlier consistent state.
Several researchers\textsuperscript{16,21} address an important issue of compressing the event log to reduce the storage requirements. A system by Chung \textit{et al.}\textsuperscript{21} focuses on enabling the latecomer support to interoperate with different groupware systems. Our system is somewhat more narrow in that it supports the DISCIPLE system, which, however, supports collaboration with arbitrary JavaBeans applications.

5 Conclusions and Future Work

Distributed collaborative systems must allow dynamic joining and leaving of sessions and therefore must provide latecomer and crash recovery support. The solution presented here offers a dynamic tradeoff between performance scalability on one hand and the cost constraints on the other hand. The system can dynamically reconfigure from a centralized architecture to a completely distributed system. Performance comparison of the algorithms clearly prefers state exporting to event logging. We are currently exploring combinations of these algorithms to leverage their individual strengths. Our architecture performs well on the Internet, allowing geographically dispersed people to work together on common tasks.

Our continuing work focuses on semantic compression of logged events. A particularly promising direction is based on integrating XML (Extensible Markup Language) as a medium for information exchange. A single data structure, the tree, imposed by XML parsers simplifies the compression process significantly. Another research direction is intelligent methods for run-time selection of the best algorithm for latecomer/crash recovery support, depending on the application-specific requirements and capabilities.

Acknowledgments

This research is supported by DARPA Contract No. N66001-96-C-8510, NSF KDI Contract No. IIS-98-72995 and by the Rutgers Center for Advanced Information Processing (CAIP).

References


