

Latecomer and Crash Recovery Support in Fault-Tolerant Groupware

Mihail Ionescu and Ivan Marsic, *Rutgers University*

In a real-time collaborative system, one user's actions must be instantaneously propagated to all the other current participants. A session might start with some users, others might join in later, and still others might leave and join again. Unlike stateless collaborative applications such as videoconferencing and discussion groups, state-full collaborative applications must offer latecomer and crash recovery support. Such applications include shared electronic whiteboards, chat rooms, collaborative design, collaborative virtual worlds, and multiplayer games.

There are two main research directions in this area: logging and playback of events, and exporting the application state to newcomers. Both have advantages and shortcomings. Event logging and playback imposes minimal requirements on applications and lets latecomers see how the existing participants reached the present state. However, it can require an excessive amount of storage, because sessions can last a long time. Bringing a latecomer up to date might also take a long time, for the same reason. The other technique, exporting the application state to each latecomer, has minimal storage requirements and takes relatively little time. However, the latecomer cannot see how the present situation developed. Also, the application must provide for exporting its data structures. The "Related Work" sidebar points to other research in this area.

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 needing little latecomer updating time;
- economical use of resources, particularly of storage; and
- flexibility in history review, from the entire history to the final state only.

The system presented here meets these requirements by implementing two alternative algorithms for latecomer support. Our solution

- distributes latecomer support across multiple hosts, thus providing fault tolerance;
- guarantees correct updating of newcomers in a finite time, with minimal system constraints;
- selects the algorithm for latecomer and crash recovery support at runtime; and
- adjusts in a simple way the number of sites involved in latecomer support to maintain a good balance between performance and cost in terms of resources.

We implemented and evaluated this solution in our framework for synchronous collaboration, called Disciple.¹ The system supports the building and sharing of collaborative applications from single-user beans (JavaBeans-compliant components, applets, and applications) and can also run as a Java applet in Web browsers.

Collaboration framework

A collaborative configuration is defined as a structure that comprises the active sites, $\Gamma = \langle S_{\alpha 1}, \dots, S_{\alpha n} \rangle$. Each site hosts an application that communicates with the applications residing at remote sites. We can define the application state in two ways: as a pair $S = \langle id, O \rangle$, where id is a unique identifier associated with the site (for instance, its IP address) and O is an ordered set of

instances of operations $O_{p1}, O_{p2}, \dots, O_{pm}$ along with the input parameters; or 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 the following holds for any two sites $\alpha, \beta \in \Gamma$: If at any time the system were to become quiescent with no messages in transit, their application states must be the same, $S_\alpha = S_\beta$. A key issue in distributed frameworks is maintaining the configuration's correctness in the presence of concurrent access.

Total ordering of events

Our system transmits events reliably but does not guarantee a particular order of arrival. Because the operations defined by applications generally do not commute, events received out of order can adversely affect the configuration correctness.

Given the events e_α and e_β , generated at the sites α and β , then e_α precedes e_β if and only if $\alpha = \beta$ and e_α was generated before e_β , or

$\alpha \neq \beta$ and the execution of the e_α 's operation at β happened before the generation of e_β .²

We define the precedence property as follows: if an event e_α precedes an event e_β , e_α executes before e_β at every site.

The precedence property does not guarantee the configuration's correctness.³ One possible solution adopted in Disciple is to extend the partial ordering offered by the precedence property to a total ordering. Given a configuration Γ with each site α sending its own events $e_{\alpha1}, e_{\alpha2}, \dots, e_{\alpha k}$, total ordering exists if every site receives all the events in the same order. We say that time t_α and time t_β at the sites α and β are logically equivalent if α receives an event e at local time t_α and β receives the same event at local time t_β , assuming total ordering of the events.

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

Internet implementation

True multicast is not possible in today's Internet, because 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.^{4,5} The first site in a configuration becomes the *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. Asymmetric total ordering is simpler to implement than the symmetric one, but it is not scalable. However, collaborative editing sessions typically do not involve more than five to 10 people, especially for the applications that we are concerned with.

The following algorithms are based on the two general approaches for latecomer and crash recovery support.

Event logging and replay

In this algorithm, each site keeps a log of all the received events that change its application state, defined as $S = \langle id, O \rangle$. 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 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 site β , and receives all the events that were logged in β 's log (note that the point-to-point connection does not participate in the total ordering process). After β sends all the events in its log, it sends a termination message to the latecomer.

Data structures

A number of data structures support event logging and replay.

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

Each site α maintains an *event log* L_α 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. Each site also maintains an ordered *late-event log* LL_α for backing up the events received while it is updating a latecomer.

Each site α maintains a *framework state* S_α (different from the application state): either *late*, *late_wait*, (being) *updated*, *normal*, *normal_wait*, or *updating*, as illustrated in Figure 1. Each site is in the *late* state at startup and transitions to *normal* after it has received and executed all the events logged at another site. In the *normal* state, the user interface is enabled for interaction. The

site visits the *updating* and *normal_wait* states when it sends the events from its private log to a latecomer.

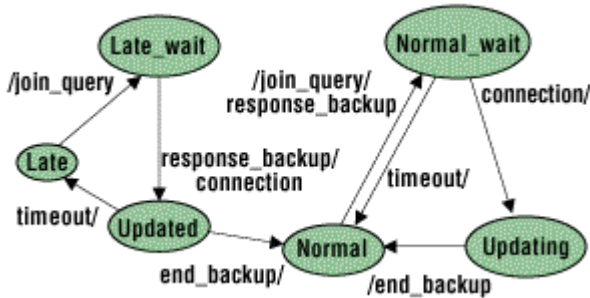


Figure 1. The state diagram for a site in a distributed collaboration system.

Each site α maintains a *query sequence number* q_α used to prevent the latecomer’s “starvation.” Assume a configuration with three sites α , β , and γ . Assume a new site δ wants to join and α is the first to respond to the *join_query* event, so α initiates point-to-point communication with δ . Suppose that α crashes immediately upon initiating the connection, obeying the fail-stop model. (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, δ resends the *join_query* event. Suppose that in the meantime α recovered (by having a high-speed connection with β , for example), and again α is first to respond to δ but crashes after a short time. If this persists, δ will starve and will never be able to reach the shared state—even if it could do so by establishing a connection with γ , for example. We solve this by adding q_α to each *join_query* event. A site can respond to the *join_query* event only if it has received all the preceding *join_query* events.

Each site maintains a *query map* M_α containing the query numbers of the current latecomers. It can obtain the query number from a *join_query* event e using the method $qn(e)$. We say that a site that responds to the *join_query* event sent by β is a *responding* site with respect to β .

The remaining configuration members might not be able to respond to *join_query*, because they did not receive all the preceding messages. Consider an example with two sites α and β and a site γ trying to join. Assume that α responds to the first *join_query* and fails during the updating. The site γ resends *join_query*, and β responds but also fails during the updating. When γ sends the next *join_query* event, no site will be able to respond even if α and β recovered and are active members. The site γ 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

Figure 2 gives a more precise specification of the algorithm executed at an arbitrary site α .

```

Lα ← empty
Sα ← late
qα ← 0
Mα ← empty
while (still active) do
  switch (Sα)
  case late:
    join:
      create a join_query event with
        the current qα and broadcast it
      Sα ← late_wait
    case late_wait:
      wait for response r
      switch (r)
      case response_backup:
        δ ← id(r)
        send a connection event to site δ
  
```

```

        Sα ← updated
    case timeout:
        if (qα > 0)
            qα ← 0
            Sα ← late
            goto join
        else
            Sα ← normal
        end
1   case updated:      // being updated
    do forever
        wait for event m from site δ
        if (timeout)
            Lα ← empty
            Sα ← late
            goto join
            qα ← qα + 1
            if (m is end_backup)
                break;
            data ← data(m)
            ← label(m)
            Lα ← Lα + <δ, data, _ℓ>
            execute the m's operation
        od
2   Sα ← normal
3   case normal:
    while (Sα == normal) do
        receive event m
        _ℓ ← label(m)
        δ ← id(m)
        switch (_ℓ) // label value
            case 1:
                Lα ← Lα + m
                if ((==α) // local event
                    broadcast m to all other sites
                    for each e in LLα
                        execute the e's operation
                    end for
4   execute the m's operation
                case 3:
                    if (m == join_query)
                        q ( qn(m)
                        find qβ in Mα for site β
                        if (q ≤ qβ+1)
                            send response_backup to δ
                            increment qβ
                            Sα ← normal_wait
                        fi
                    fi
                end switch
            od
    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
5 case updating:
   $e = \text{getNext}(L_\alpha)$ 
  while ( $e \neq \text{null}$ ) do
    send  $e$  to site  $\delta$ 
    receive event  $m$ ;
    if (!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 end_backup to site  $\delta$ 
     $S_\alpha \leftarrow \text{normal}$ 
6   end
   od

```

Figure 2. Specifications of the event logging and replay algorithm.

The algorithm is completely distributed and resilient to site failures. However, each site must keep an event log in memory (or on disk). This might be undesirable, because some hosts could have lower capabilities. We solve this problem by letting the user decide at runtime whether his or her site should keep the log or not. If the site does not maintain the event log, it lacks the *updating* state and ignores the join events. In this case, only a subset of sites support latecomers.

Any number of sites can be involved. We found the “centralized” architecture suitable for an environment comprising one high-end server (called a *central site*) and multiple low-end computers. However, a crash of the central site completely eliminates 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 join events before it is activated. It listens to the multicast channel and logs all the events. This technique resembles passive replication,⁶ except that our 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 2). Every site can at a given quantum of time determine which sites left or joined the session. Also, each site maintains the server id as the id of the central site.

When a site discovers that the central site left the session, it broadcasts a special event, *query_server*. 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 server id. The backup server now becomes the central site and starts responding to the join events. The program blocks all user actions from the time a site discovers that the central site left the session until it receives the *server_id* event.

In an asynchronous environment like the Internet, differentiating between slow and failed members is impossible.⁷ Our failure detection mechanism is based on the assumption that if a site does not send the heartbeat for a certain number of periods, the site crashed.⁸ 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 it is the first one. However, there might be other members that could not answer because of network partitions. Supporting partitioned networks and merging corresponding states is one direction in our continuing work.

If the set of involved sites is small, the updating process might take longer if many latecomers try to join simultaneously. This trade-off between performance and cost is characteristic for distributed systems. Our framework offers the flexibility to dynamically increase the number of involved sites to achieve the right balance.

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 might not

always be correct. The latecomer could receive some new events before the `end_backup` event, and the updating site could receive a new event after sending an `end_backup` event (see 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 sometime during the backup. These correspond to two (possibly different) physical times, t_α at the latecomer's site and t_β at the updating site. The updating site stops sending backup events if it notices that it received the next event in the backup log after t_α , using the timestamps from the event log. At the same time, the latecomer buffers the new events received after t_α at the multicast channel (that is, independently of the point-to-point connection); it processes these events after it receives the `end_backup` event. In our implementation, we use the `join_query` event to determine logically equivalent times. (The latecomer too receives its own `join_query` event, because the event is sent to the multicast channel, which is totally ordered.)

A sketch of the event logging and replay algorithm's proof of correctness is presented in a sidebar.

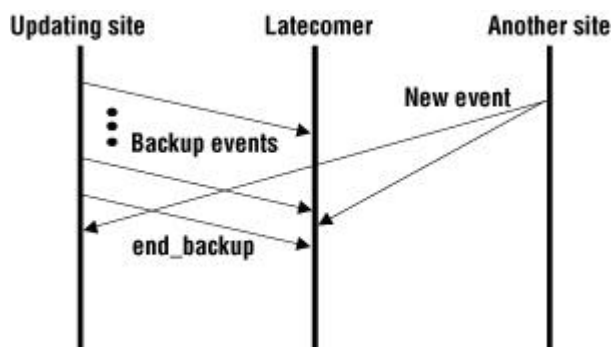


Figure 3. Example of the latecomer ignoring a new event.

Crash recovery

Crash recovery is based on latecomer support. In addition, the system attaches an increasing sequence number 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 stable storage along with the current sequence number. As mentioned earlier, we assume that sites crash obeying the fail-stop model and the network is reliable—that is, it will not partition. When a site wants to rejoin the configuration, it loads the events first from 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 lets the latecomer replay all the events to see the history of the collaborative work. However, the algorithm includes a continuously growing data structure—the event log—so it might be 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 either will not be able to do anything during this period or, if multithreading is used, will only be able to respond slowly.

The second algorithm remedies these problems.

Exporting the application state

In this algorithm, we define the application state at site α as $S = \langle \alpha, o \rangle$, where o is an array of bits representing the serialized application state. The main idea 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 framework encapsulates the array in a predefined type of event and sends it to the latecomer.

The basic structure of this algorithm is similar to the previous one, so Figure 4 outlines only the main differences. The blocks between the numbered lines change accordingly.

```

    if (timeout)
        goto join
    execute the event's operation
2    $S_\alpha \leftarrow normal$ 
3   case normal:
        while ( $S_\alpha == normal$ ) do
            receive event  $m$ 
                 $\leftarrow label(m)$ 
             $\delta \leftarrow id(m)$ 
            switch ( )
                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:
5   do in parallel:
        obtain serialized application state
        ||
        receive event  $m$ 
        if (!timeout)
            if ( $label(m) == 1$ )
                 $LL_\alpha \leftarrow LL_\alpha + m$ 
        od
        send serialize event to site  $\delta$ 
         $e = getNext(LL_\alpha)$ 
        while ( $e != null$ ) do
            send  $e$  to site  $\delta$ 
            receive event  $m$ 
            if (!timeout)
                if ( $label(m) = 1$ )
                     $LL_\alpha \leftarrow LL_\alpha + m$ 
                 $e = getNext(LL_\alpha)$ 
        od
6   send end_backup to site  $\delta$ 

```

Figure 4. The algorithm for exporting the application state.

Serializing the application state can take a significant amount of time for a complex application state. During this time, other sites might 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 algorithm's main advantage is that the sites need not maintain an event log. Often there is no reason to do so, because the latecomer is not interested in the session history, but a problem arises if it is interested. Also, this algorithm loses the previous algorithm's generality in the sense that the application must be aware of the latecomer support (it has to implement the framework-specific interface).

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

Implementation and performance

We evaluated the Disciple framework's performance using both algorithms, in a LAN environment and in the global Internet. We used two collaborative applications.

The first was a medical imaging application, with a large image database and a real-time microscope, for interactively indexing and retrieving pathology images.⁹ The application analyzes the color, shape, and texture of blood-cell images and detects and identifies leukemia cells by comparing them to a database of diagnosed cases. Here, the session history is important for the latecomer's understanding of the current session state, as shown in Figure 5. Replaying all the events seems to be the only feasible solution. So, this application does not implement the second algorithm's latecomer support interface.

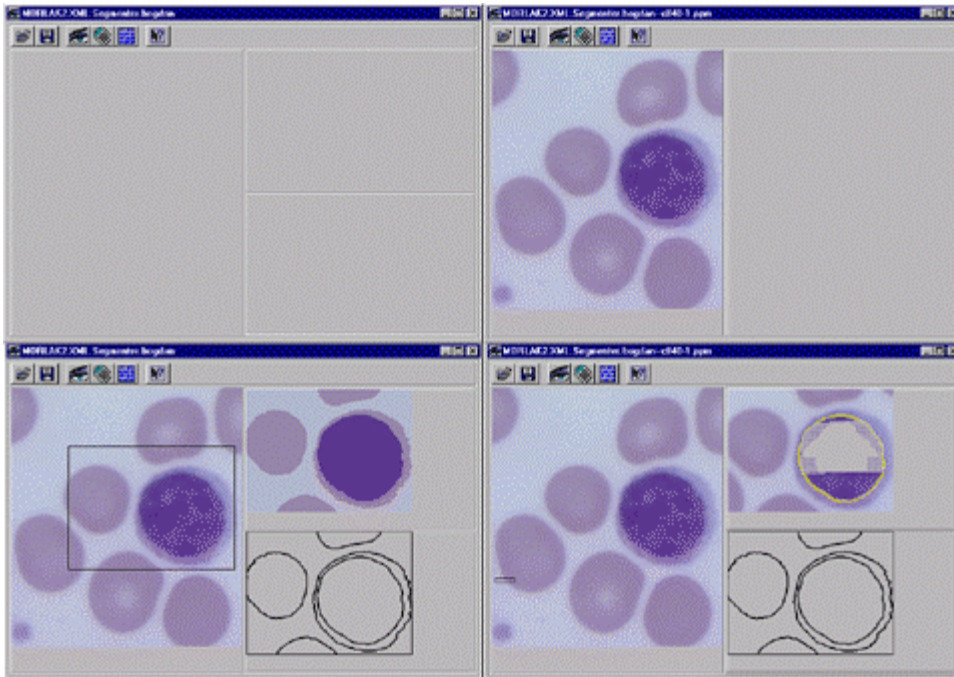


Figure 5. The medical imaging application and some of the image analysis steps seen by a latecomer.

The second application was a classic shared whiteboard, in which multiple users collaborated to create a complex 2D drawing. It can use either algorithm. A latecomer might be interested in replaying all the events or going directly to the final whiteboard state. We choose at runtime what algorithm to use as follows. As mentioned earlier, Disciple determines if the application implements the latecomer interface. If it does, Disciple lets the user choose which algorithm to use to support the latecomers.

Let's assume that during a collaborative session the shared application state becomes as shown in the upper row in Figure 6. If we choose the second algorithm, only the final application state will be displayed when a latecomer joins the session. If we use the first algorithm, Disciple will replay the sequence of all events.

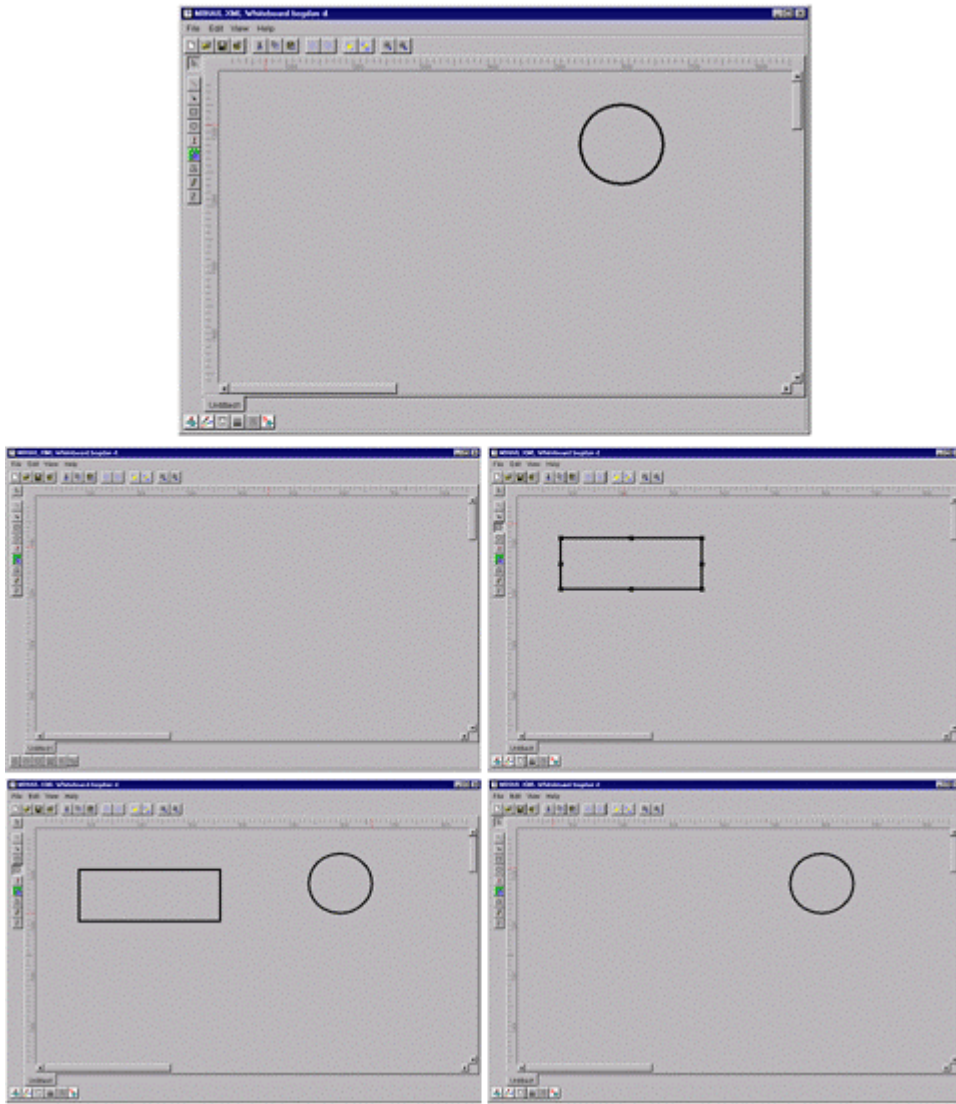


Figure 6. The whiteboard application at the start (top) and the intermediate stages of collaboration.

Performance in a LAN environment

We used the whiteboard as the base application to compare the two algorithms' overall performance. For simplicity, we assume that the whiteboard can handle a single type of object (for example, the rectangle) and defines four operations on the object: Create, Transform (translate, rotate, scale), ModifyAttribute (color, line width, and so on), 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_{op} (100 in this case), of which the number of Create operations varies from 1 to N_{op} . If $N_{Create} < N_{op}$, the remaining operations are randomly distributed between Transform, ModifyAttribute, and Delete. We define the *update time* as the time required to bring a latecomer up to date after it issues a $join_query$ event. The system plays the events immediately, one after another, rather than trying to simulate real time as in systems that additionally record speech conversation.¹⁰ Figure 7 gives the average results for 10 executions of the test scenario.

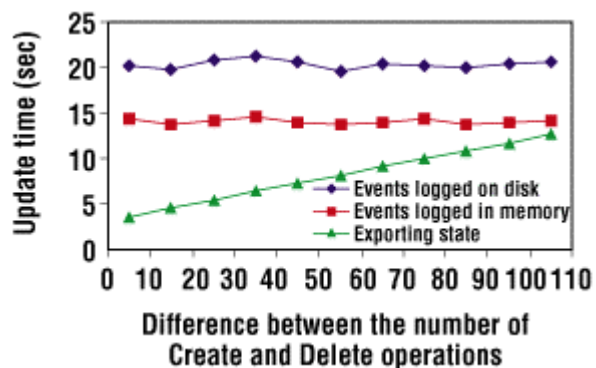


Figure 7. Update time for the two algorithms with events logged on disk and in memory.

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

Performance on the Internet

We implemented the medical imaging application as a Java applet, as shown in Figure 8. In this configuration, the centralizer and the Web server run on the same host.

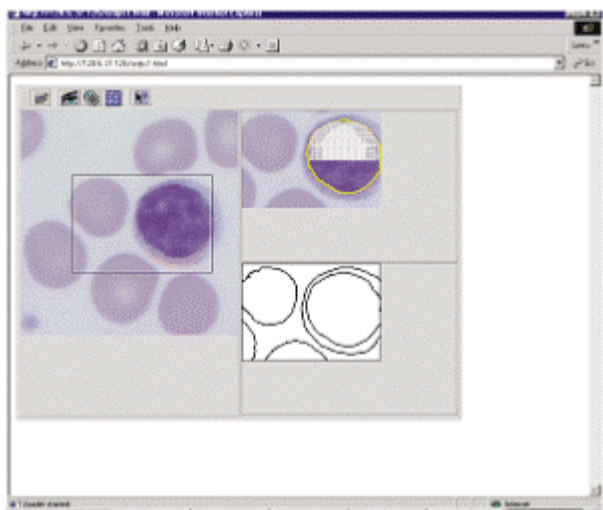


Figure 8. The medical imaging applet viewed in an Internet Explorer window.

We are mainly interested in this centralized scheme's scalability. 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. Figure 9 presents the average results.

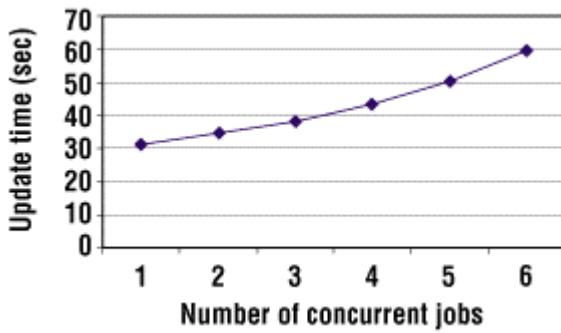


Figure 9. Update time for the event logging algorithm in an Internet environment.

System performance 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 four latecomers). This is because of hardware limitations of the Web server, because each connection requires a separate thread.

The solution presented here offers a dynamic trade-off between performance scalability and cost constraint. The system can dynamically reconfigure from a centralized architecture to a completely distributed system. In comparing the algorithms' performance, we found that state exporting was clearly better than 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 the 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 runtime selection of the best algorithm for latecomer and crash recovery support, depending on application-specific requirements and capabilities.

Acknowledgments

DARPA contract no. N66001-96-C-8510, the US National Science Foundation KDI contract no. IIS-98-72995, and the Rutgers Center for Advanced Information Processing support this research.

References

1. I. Marsic, "DISCIPLE: A Framework for Multimodal Collaboration in Heterogeneous Environments," *ACM Comp. Surveys*, vol. 31, no. 2, electronic supplement, June 1999.
2. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, vol. 21, no. 7, July 1978, pp. 558–565.
3. C. Sun et al., "Achieving Convergence, Causality-Preservation, and Intention-Preservation in Real-Time Cooperative Systems," *ACM Trans. Computer-Human Interaction*, vol. 5, no. 1, 1998, pp. 63–108.
4. P.D. Ezhilchelvan, R.A. Macedo, and S.K. Shrivastava, "Newtop: A Fault-Tolerant Group Communication Protocol," *Proc. Int'l Conf. Distributed Computer Systems (ICDCS '98)*, IEEE CS Press, Los Alamitos, Calif., May 1998, pp. 202–209.
5. H. Garcia-Molina and A. Spauster, "Ordered and Reliable Multicast Communication," *ACM Trans. Computer Systems*, vol. 9, no. 3, Aug. 1991, pp. 242–271.
6. R. Gueroui and A. Schipper, "Software-Based Replication for Fault Tolerance," *Computer*, vol. 30, no. 4, Apr. 1997, pp. 68–74.
7. M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, vol. 32, no. 2, Apr. 1985, pp. 374–382.
8. K.P. Birman and T.A. Joseph, "Exploiting Virtual Synchrony in Distributed Systems," *Proc. 11th ACM Symp. Operating Systems Principles*, ACM, New York, Nov. 1987, pp. 123–138.
9. D. Comaniciu, P. Meer, and D. Foran, "Image Guided Decision Support System for Pathology," *Machine Vision and Applications*, vol. 11, no. 4, Dec. 1999, pp. 213–224.

10. N.R. Manohar and A. Prakash, "The Session Capture and Replay Paradigm for Asynchronous Collaboration," *Proc. European Conf. Computer Supported Cooperative Work*, Kluwer Academic Publishers, Dordrecht, the Netherlands, Sept. 1995, pp. 149–164.



Mihail Ionescu is a PhD candidate in computer science at Rutgers University and a faculty member in the Computer Science Department at Politechnica University of Bucharest. His research interests include groupware and mobile computing, and he is responsible for communication and concurrency control in the Disciple system. He received his BS in computer science from Politechnica University of Bucharest, Romania. Contact him at the Center for Advanced Information Processing, Rutgers Univ., 96 Frelinghuysen Rd., Piscataway, NJ 08854-8088; mihaii@caip.rutgers.edu.



Ivan Marsic is an assistant professor of electrical and computer engineering at Rutgers University. He is the chief architect of the Disciple system, a groupware system that enables teams to collaboratively access, manipulate, analyze, and evaluate multimedia data using geographically distributed, networked information systems. His research interests include groupware, mobile computing, computer networks, and human–computer interfaces. He received his BS and MS in computer engineering from the University of Zagreb, Croatia, and his PhD in biomedical engineering from Rutgers University. Contact him at the Center for Advanced Information Processing, Rutgers Univ., 96 Frelinghuysen Rd., Piscataway, NJ 08854-8088; marsic@caip.rutgers.edu.

Related Work

Research efforts in fault tolerance and failure recovery have a long history in the distributed systems community. These issues have received the most attention in the distributed databases field. Fault tolerance methods using message logging and checkpointing are particularly interesting because they are general-purpose and do not require excessive amounts of storage. (Checkpointing is a technique in which, during normal execution, application states are periodically saved in stable storage and used during recovery for rollback to an earlier consistent state.) Examples of such techniques are dependency tracking during rollback to a common state¹ and using transitive and indirect dependencies to obtain the most recent consistent global state after recovery.² Virtual synchrony^{3,4} provides a formal definition of group communication in the presence of failures by offering a well-defined guarantee regarding the order of messages and of failure notifications. Other systems handle member failure during state updates.^{5,6}

Crash recovery schemes for general distributed systems offer a basis for groupware latecomer and crash recovery schemes. The event-logging scheme derives from the general message-logging scheme. However, straightforward application is not possible due to the particularities of groupware systems, in which users are mainly interested in system responsiveness and in maintaining the correctness of the configuration. The rollback algorithms are useless because of the confusion they introduce to users. Also, the events transmitted in the network are usually small, so the total size of the event log might not be excessive.

The architecture of the groupware system plays an important role in designing latecomer support. Most of the systems can be classified into centralized^{7–9} versus replicated architectures.^{10–12} 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 (for example, in Suite⁸). Most solutions take either the event-logging and replaying approach^{7,9,10} or application state exporting,^{8,11,12} but not both or their combination. In some cases,^{9,11,12} the latecomer support module is centralized, although the overall system architecture is replicated. In GroupKit, a dedicated server provides the latecomer service.¹¹ As with any centralized system, a server failure affects the whole configuration. Corona⁹ offers fault tolerance by maintaining logs in the server's stable storage, as well as on the clients (in case the server crashes). Most of these systems do not tolerate failures during latecomer updating.

One system based on event logging provides a VCR-like interface, so that the late user can pause, skip, and fast-forward the replay.¹⁰

Several researchers address the important issue of compressing the event log to reduce storage requirements.^{7,13}

One of these systems also focuses on enabling latecomer support so that it can interoperate with different groupware systems.¹³ Our system is somewhat more narrow in that it specifically supports the Disciple system, which in turn supports collaboration with arbitrary JavaBeans applications.

References

1. R. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Trans. Computer Systems*, vol. 3, no. 3, Aug. 1985, pp.

204–226.

2. P. Sistla and J. Welch, “Efficient Distributed Recovery Using Message Logging,” *Proc. 8th Ann. ACM Symp. Principles Distributed Computing*, ACM, New York, Aug. 1989, pp. 223–238.
3. K.P. Birman and T.A. Joseph, “Exploiting Virtual Synchrony in Distributed Systems,” *Proc. 11th ACM Symp. Operating Systems Principles*, ACM, New York, Nov. 1987, pp. 123–138.
4. L.E. Moser et al., “Extended Virtual Synchrony,” *Proc. Int’l Conf. Distributed Computer Systems (ICDCS ’94)*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 56–65.
5. M. Cukier et al., “AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects,” *Proc. 17th IEEE Symp. Reliable Distributed Computing*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 245–253.
6. P. Narasimhan, L. Moser, and P.M. Melliar-Smith, “Replica Consistency of CORBA Objects in Partitionable Distributed Systems,” *Distributed Systems Eng.*, vol. 4, no. 3, Sept. 1997, pp. 139–150.
7. G. Chung, K. Jeffay, and H. Abdel-Wahab, “Accommodating Latecomers in Shared Window Systems,” *Computer*, vol. 26, no. 1, Jan. 1993, pp. 72–74.
8. P. Dewan and R. Choudhary, “Coupling the User Interface of a Multiuser Program,” *ACM Trans. Computer–Human Interaction*, vol. 2, no. 1, Mar. 1995, pp. 1–39.
9. H.S. Shim and A. Prakash, “Tolerant Client and Communication Failures in Distributed Groupware Systems,” *Proc. 17th IEEE Symp. Reliable Distributed Systems*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 221–227.
10. N.R. Manohar and A. Prakash, “The Session Capture and Replay Paradigm for Asynchronous Collaboration,” *Proc. European Conf. Computer Supported Cooperative Work*, Kluwer Academic Publishers, Dordrecht, the Netherlands, Sept. 1995, pp. 149–164.
11. M. Roseman and S. Greenberg, “Building Real-Time Groupware with GroupKit, a Groupware Toolkit,” *ACM Trans. Computer–Human Interaction*, vol. 3, no. 1, Mar. 1996, pp. 66–106.
12. A. Chabert et al., “Java Object-Sharing in Habanero,” *Comm. ACM*, vol. 41, no. 6, June 1998, pp. 69–76.
13. G. Chung, P. Dewan, and S. Rajaram, “Generic and Composable Latecomer Accommodation Service for Centralized Shared Systems,” *Proc. IFIP Working Conf. Eng. for Human–Computer Interaction*, Kluwer Academic Publishers, Dordrecht, the Netherlands, Sept. 1998, pp. 129–148.

Proof of Correctness

The event logging and replay algorithm works for the general case in which a latecomer joins a correct configuration, but the proof is complex. So, for the sake of simplicity, we will assume that no messages are in transit when the latecomer joins. We also make the following assumptions about the communication medium:

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

Consider having a correct configuration $\Gamma = \langle \alpha_1, \dots, \alpha_n \rangle$ at time t_0 and a latecomer, site β , wants to join.

Lemma 1

Consider a subset $A \subseteq \Gamma$ at time t_0 , where $|A| = C$, $A = \{\alpha \in \Gamma \mid \alpha \text{ is involved and } \alpha \text{ is responding with respect to } \beta\}$. Obviously, $C \leq n$, and we assume 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 (because Γ is a correct configuration at that time), denoted by $E_0 = \{e_1, e_2, \dots, e_p\}$, where p is the number of events the active sites generate until t_0 . Then, for every site $\tau \in A$, $L_\tau = E_0$.

Proof. The proof follows from the definition of configuration correctness. Consider an arbitrary site $\tau \in A$. Because $A \subseteq \Gamma$, this implies $\tau \in \Gamma$, so all the events from E_0 were executed at τ . But, according to the algorithm, each executed event is also inserted in the log L_τ 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 β joins Γ at time t_0 , after a finite time all the events from E_0 will be executed in the order e_1, e_2, \dots, e_p at β .

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

Assume now that τ fails before sending all the events from L_τ . 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. Because 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 τ will no longer respond 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 Γ 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 $E_1 = \{e_1, e_2, \dots, e_q\}$ in the same way that we defined the set E_0 , where $q \geq p$. Let us denote the set $(E_1 - E_0)$ by E . The latecomer buffers the events received from E before the latecomer has received all the events from E_0 . Γ is correct at time t_1 , so no messages are in transit. This implies that the latecomer receives all the events from E_1 at time t_1 . This proves the theorem.