Scalable Asynchronous Interaction Based on Selective Recording and Replaying of X-Protocol Streams

Lassaâd Gannoun and Jacques Labetoulle
Institut Eurécom
2229, route des Crêtes
F-06904 Sophia-Antipolis
{gannoun, labetoul}@eurecom.fr

Abstract

This paper deals with the problem of allowing a user to record and replay a specific portion of a window application session in order to build an efficient and scalable asynchronous interaction. Our approach is based on a method that enables a latecomer to dynamically join a conference and share applications used in the conference. We propose to extend this method by archiving the state modifications to the window system as consistent revival points that can be stored with the user's recorded application session. Later, in a replaying step we can impose a specific revival point to a remote window system and start replaying from this point. We propose two mechanisms applied in the recording step to allow later the displaying of the contents of all windows generated by the recorded application. The first is based on simulated window system events, and the second applys window system requests to get and store the contents of all application's windows contents.

Keywords: distance learning, distributed systems, X-protocol, application recording/replaying session.

1 Introduction and motivation

Shared window systems allow multiple users, each on their own workstation, to view and interact with a single user application [1],[2],[5],[6],[7]. Shared window systems have a major limitation, they are synchronous and thus work only when all participants are on-line at the same time. Application sharing would be even more effective if efficient collaboration could be extended to asynchronous interaction that does not require all participants to be on-line at the same time.

This requirement is the origin of designing and implementing an asynchronous interaction method based on an asynchronous application sharing service [4]. The asynchronous application sharing service is based on two complementary services. The first, is the application recording service which allows a user (e.g. student) to record his X-application session and to add comments along all his/her session. The second is the application replaying service enabling a remote partner user (e.g. professor) to replay the recorded X-session and to understand the partner user's work (e.g. student). A major limitation of those services is that they provide sequential recording and replaying of X-window streams.

This problem can be solved by designing a method for a direct access on an X-window stream. This method is basically based on storing a persistent revival point and later, accessing directly the X-window stream from this stored revival point. This method restores the stored point and imposes it to the window system server (i.e. X-server). Hence replaying of the remainder X-window stream can be started from this point.

The remainder of this paper is organized as follows: first we describe. First, we present a related work on a method for a dynamic participation in a computer based conferencing system introduced in [3]. Afterwards, we present our approach that is based on this method and that provides a direct access on a stored X-window stream. Then, we evaluate some protocols for a scalable and flexible asynchronous interaction. Finally, we provide some concluding remarks and outline our future plans.

2 A dynamic participation method in a computer based conferencing system

This method was designed to allow a latecomer to participate in a window shared session. The approach

adopted in this method is to record the modification made by requests to the resources allocated on the X-server.

2.1 X window system resources

An X application programming model presents six basic abstractions: window, cursor, graphics context, pixmap, coulormap and fonts. Windows and pixmaps are both referred as drawables. Resources are created manipulated and destroyed by the server in response to clients requests. The following is brief description of these resources.

2.2 Recording modifications to resources

This approach consists to catalogue changes a client can make to the server state. A client may change the server state as follows:

- create private resources (e.g. a client can create a set of windows, a set of colors, etc...for its use)
- change attributes of resources that it creates itself or that did not create by itself.

This approach of concentrating on modifications made to resources guarantees that a minimal set of information is kept about the changes made by the client to the server state. Whenever a new resource is created by a client, data structures are created to record the attributes of the resource. When the clients change the attributes of a resource, the data structures associated with this resource will be modified. When a client sends a request to free the resource, the data structures associated will be deleted.

A problem arises when immediatly deleting a resource that the client frees. Then we should not delete a resource that is required by another resource which is not deleted. To avoid this problem, this method apply an algorithm that creates and maintains a dependency relationships between resources. Figure 1 shows the dependency graph that result from the CreateCursor request.

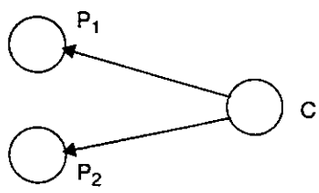

Figure 1. CreateCursor request dependency graph.

When a request to free a resource R is encountered, the algorithm checks if any resource depends on R. Only when no other resource depends on R can R's information be deleted from maintained data structure. After recording modifications to resources and maintaining a dependency relationship graph between resources, using this produced graph we can modify the state of a latecomer's server.

In the next section we show how we can adapt this method to use it in asynchronous manner to allow direct access on an X-window stored session.

3 An approach for a direct access to an X-protocol stream.

We propose to apply the method described in the previous section to access directly a stored X-protocol stream. We know that it is not possible to access directly a specific media unit (X-request) of an X-protocol stream and starts playing requests (to an X-server) from this media unit. This is because a particular media unit of an X-protocol stream is effectively dependent of a the media units preceding it, and hence of a certain context of allocated resources on the X-server. To explicit our approach, we introduce these notations:

S: an X-protocol stream with n media units

R_i: the media unit (X-request) of rank i in the stream S with $1 \le i \le n$

P_i: the state point i of the allocated resources on an X-server after playing all media units R_k in the stream S that precedes R_i with $1 \le k < i$.

To access directly a stored media unit R_i we should save the state point P_i at the recording step. This point reflects the state of the server before playing the X-request R_i. Figure 2 shows the association made between state server points and X-protocol requests.

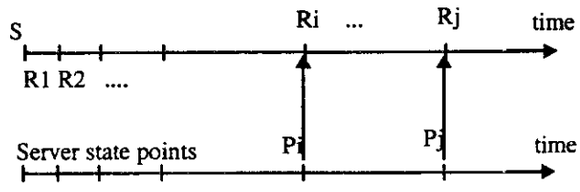

Figure 2. Relationship between X-requests and server states

In the replaying step if we desire to start replaying from the request R_i we should restore the correspondent server state point P_i and impose this point to the server, then we can start playing requests from R_i. If we apply with a "brute approach" the method described in the previous section we will not able to impose correctly a server state point in a replaying step. Because, this method only

records the modifications made by the application to the server state and assumes that displaying the current windows contents will be made by the client. However, in the replaying step we have only a stored X-protocol stream request and we have not any client running.

To cope with this problem we propose two mechanisms to allow later the displaying of all the client windows. The first mechanism is based on simulated packet events, and the second apply GetImage request and PutImage request to refresh the GUI of the client.

3.1 Simulated packet events

This mechanism is applied in the recording step. In this step we apply the method proposed by [3] to maintain a dependency relationship graph between current created resources. If we desire to save a state point P_i, then we send simulated expose events to the client being recorded. These special server events are normally sent by the server to the client to indicate that portions of a window become visible. Each expose event specifies a rectangular region inside the window that becomes visible. In response, the client will generate the appropriate requests to draw an up-to-date image on his window. These generated X-requests will be stored to further refresh the contents of all windows in the replaying step.

One would expect that to get all requests for refreshing the contents of a given window we should simulate single expose events for that window. However, it appears that is necessary in special cases to simulate multiple expose events for a given window. This depends on the current state of the visible window relative to the other overlapping windows.

To determine whether or not to generate single or multiple expose events for a window, we archive expose events sent by the server in the recording step. The archiving consists of maintaining an event data structure which represents the most recent expose events sent by the server relative to a given window. After saving a server state point, we generate expose events from these event data structure and we send them to the client. The recording agent handles the request packets sent by the application and stores them as refreshing requests.

We present in the following another approach which is based on generating GetImage requests to later refresh the contents of all client windows.

3.2 Generated GetImage requests.

In rarely cases where it is extremely hard for the client to refresh the window's image, the client can request the server to refresh the window's image, the client can request the server to do the refreshing. In this case there will no expose events sent from the server to the client and the scheme described above can not be applied. To get the current image of a window there is a request called GetImage that acquires the contents of a window image from the server.

In a recording step when we store the state of a server, GetImage requests are generated for all windows created by the client. The server will send replies as a response to these requests. Then, the contents of windows are stored. Later, in the replaying step we can use PutImage request to up-date the contents of all windows. Figure 3 shows the architecture of a recording agent.that archives the server's state.

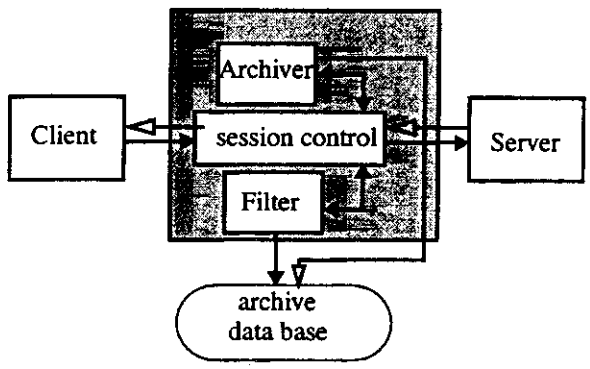

Figure 3. Architecture recording agent archiving server's state

The recording agent consists of three components: the session control, the filter and the archiver. The session control, control the recording process. It handles the X protocol traffic between the client and the server, controls the recording of X requests and the archiving of server states. The filter filters requests to be stored. The Archiver maintains a dependency graph of the resources created. This dependency graph helps to impose a specific state point P_i to a server in a replaying step.

In the next section we present different approaches using the recording of a server state point and later restoring and imposing of this state point on a new server.

4 Different approaches for a scalable asynchronous interaction

During an asynchronous interaction based on recording and later replaying of an X-window session, we have faced the problem of accessing directly a stored X-session. This problem can be solved by the following two policies.

- selective recording and replaying of an X-window stream policy

- introducing revival points on the recording of an X-window stream

In order to criticize these policies we illustrate scenarios of an asynchronous interaction between a professor and a student based on each of these policies.

An asynchronous intercation can be based on a selective recording and replaying of X-window streams. This policy is applied in two steps: the recording and the replaying step.

Within this policy, the recording agent do not have to record all X-application requests. It uses a method for dynamic participation in a computer based conferencing system[3] described in section 2 to record the modifications made by the application to the X-server state. When the student encounters a problem he comments it and the recording agent saves the X-server state as a consistent revival point as described in section 3. This allow later the replaying agent to replays requests from the stored revival point.

In the replaying step, the replaying agent has to impose the recorded revival point to the X-server. Hence, the professor can start replaying the X-window session from this revival point. This policy is interesting when the professor can understand the student's work from only this stored revival point. Applying this policy the student can store only a portion of his/her application session. Therefore, this policy has low memory and CPU requirements but presents the following limitations:

- The professor can only start replaying the stored X-session from the revival point. It may be that to understand the problem encountered by the student requires the viewing of a student's session portion that precede the stored revival point. In this case, the professor is not able to view the X-session portion preceding the stored revival point.
- The problem of a direct access to the X-window stream portion remains. For example if the professor wants to access the next student's problem that occurs one hour after the previous problem he should also wait one hour to encounter the next problem.
- Within this policy there is no way to go-fast backward or to fast forward (skipping) in order to view rapidly other portions of the real student's application session. This functionality is a key aim of an efficient replaying service, and then can be useful to understand student's session.

An alternative approach that solves these problems is the second policy that introduces revival points in the student's stored application session.

In the recording step, the recording agent records all X-window stream requests. During this step, revival points are also saved on the recorded session. To save a revival point the recording agent uses the method suggested in section IV to save a revival point and to restore and impose that point to the current X-server. We discuss later several mechanisms that can control the saving of revival points.

In the replaying step, the replaying agent replays the student's stored session sequentially. When the professor wants to access directly a specific portion, than he could do it by restoring the correspondent revival point and imposing it to the window system server (X-server). After that he starts replaying the student's recorded session from this point.

The approach adopted here is the second one because it combines the sequential and the selective recording and replaying. The more interesting feature with revival points is that we are able to go-backward and view other previous X-window session portions.

4.1 Control of saving revival points

Some relevant questions raise when saving the revival points. How we can efficiently save revival points during application session and who can control the saving process of these revival points? we distinguish three different approaches can be applied:

- User driven approach
- Time driven approach
- GUI events approach

In the user driven approach, the user running the application (e.g. student or professor) controls the saving of revival points. Then the user decides whether or not to save a certain revival point regard to the semantic context of the evolution of his application session. The saving of a revival point occurs respect to the semantic context of the user's work. When the user judge that he starts a novel step of his work than he can save a revival point which reflects. This approach do not take account of the time elapsed since the previous saved revival point.

Saving revival points can be done by computing the time elapsed since saving the last revival point. Hence, at each period of τ units of time (seconds or minutes e.g. τ=10mn) a revival point is saved. However, this approach do not take in account the application session context. Then we can obtain successive revival points saved and all of them reflect the same semantic session context.

The third approach is the GUI events that can control the saving of a specific revival point. For example, when the application opens a new window (e.g after forking a new process) then this event translates a certain semantic application context in the evolution of the user's work. This event can produce the saving of a revival point. It's obviously that an optimal control approach for saving

revival points should be a combination of all these described approaches. We give in the following an algorithm that controls the saving of revival points: we define here three basic events:

- user_control_event: user order to save a revival point
- gui_event: Graphical user interface event.
- timeout_event: this event is raised when the maximum of the time (τ units of time) since saving the last revival point is elapsed.

We define these system parameters:

- P_i: state point eligible for saving.
- R_i: number of requests generated since the last revival pointsaving.
- MaxReq: the maximum of requests generated since the last revival point.
- τ: the maximum of the inter revival point period
- Minτ: the minimum of the inter revival point period
- Timelapsed: the time since the last saving of the revival point

Control Algorithm:

```
Switch(incoming_event)
  case user_event:        Save (Pi)
  case timeout_event: if (Ri > MaxReq)
                          Save(Pi)
  case gui_event: if (Timelapsed > Minτ)
                          Save(Pi)
End Switch
```

This algorithm gives a priority to the user to save a revival points when he decides to do this. The elapsed time driven approach is constrained by the number of generated requests since the last saving of a revival point. This prevents the saving of insignificant revival point after a silence period (after a pause). The GUI interface event approach is also constrained by the time elapsed from the last saving of a revival point, because if we have just saved a revival point, than it is not necessarily to save another one even if we receive a GUI event.

5 Conclusion

In this paper we improve an asynchronous interaction method by provinding a novel method for a selective recording and replaying of stored X-window streams. Our proposed approach is based on a method for accommodating latecomers in a computer based conferencing system. This method is extended by two mechanisms to allow saving of consistent revival points on a stored X-window session. The first mechanism is based on simulated server events which allows to get from the client a series of requests that help restoration of the GUI of a client in a replaying step. The second is based on generated requests that gets from the server and after stores the contents of all windows being displayed.

Finally we propose an approach for a scalable asynchronous interaction based on combining the sequential and selective recording. In this approach we record revival points on a stored X-session. This helps to have a flexible replaying of large stored X-window streams. Finally, recording of revival points is governed by a control algorithm which gives to the user the high priority to save at any time a revival point.

In the future we plan to design a method for a rapid switching from a revival point to another one. This is interesting when we search to optimize time processing for an optimal switching inter-revival point.

6 References

[1] H. M. Abdel-Wahab and M. A. Feit, "XTV: A framework for sharing X-window clients in remote synchronous collaboration", In *Proceedings of the IEEE Conference on Communications Software : Communications for Distributed Applications and Systems* 1991 pp. 159-167. Chapel Hill, North Carolina.

[2] J. Baldeschweiler, T. Gutekunst and B. Plattner, "A survey of X protocol multiplexor", *ACM SIGCOM* 1993, (pp 16-24).

[3] G. Chung, K. Jeffay and H. M. Abdel-Wahab, "Accommodating late-comers in shared window system", *IEEE Computer, 26(1)* pp. 72-74.

[4] L. Gannoun, Ph. Dubois and J. Labetoulle, "Asynchronous Interaction Method for a Remote Teleteaching Session", International Journal of Educational Telecommunications 1997, 1(3) pp. 41 - 59.

[5] T. Gutekunst, D. Bauer, G. Caronni, Hasan and B. Plattner, " A distributed and policy-free general-purpose shared window system", *IEEE/ACM Transactions on Networking.*

[6] J. C Lauwers and K. A. Lantz, "Collaboration awareness in support of collaboration transparency: requirements of the next generation of shared window systems", In *Proceedings of the ACM CHI'90 Conference (Human Factors in Computing Systems)*, pp. 303 -311. Seattle.

[7] G. McFarlane, "Xmux-a System for computer supported collaborative work", *In Proceedings of the 1st Australian Multi-Media Communications, Applications and Technology Workshop.* pp. 12-28. Sydney 1991.