A Generic Shared Window Architecture and Some Issues

Chee-Wen Shiah, and Wen-Chin Chen

Communication and Multimedia Lab.
Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan, R.O.C

Abstract

The *shared workspace* had been considered as a significant feature of the synchronous concurrent engineering environment. An efficient way to provide shared workspace is the *shared window system* (or *shared application system*) which lay between the applications been shared and the underlying window management system (WMS). With shared window system, the single-user applications can be transparently shared among multiple participants without any modification under collaboration environment. There are three different paradigms to implement a shared window system: *event-sharing* paradigm, *UI-image-sharing* paradigm, and *request-sharing* paradigm. This paper will discuss the characteristic and implementation differences among these three sharing paradigms, and then propose a *generic shared window system architecture* to embody all of them in single architecture. Two crucial function groups: sharing activity management and I/O redirection/reproduction, will be separately designed within this generic architecture.

Several critical implementation problems such as: *latecomer problem*, *application spontaneous sharing problem*, and *cross-platform sharing problem*, will also be discussed. Finally, an application recorder/player system using request-sharing paradigm was proposed to illustrate the utilization of application sharing technique on non-CSCW scenario.

1 Background

The shared workspace, which provides an identical visual and operable working area among geographically separated participants, had been proved to be the most important feature of a synchronous collaboration system.[] There are two basic approaches to achieve the objective of shared workspace. The first, called *collaborative-aware* approach, relies on the development of so-called collaborative-aware applications, which directly support multiple, simultaneously active user input, and/or synchronize the output to all participants.[] The second, called *collaborative-*

transparent approach, introduces some mediate layers between existing single-user applications and underlying window management system (WMS) to support multiple user operations and output distribution, and makes the applications transparently sharable.[] The greatest drawback of the collaboration-aware approach is that all applications been shared must be specially redesigned from scratch. To leverage the large base of existing single-user applications, we adopt the collaborative-transparent approach to build our shared window system.

Before starting our discussion, the model of interaction between applications and WMS should be understood. Most modern WMSs (such as: X-window, MS Windows, OS/2, Presentation Manager, Macintosh System.). employ similar concept: concealing the peripheral hardware detail, providing the applications a high-level interface to perform graphical output, and using message-passing mechanism to communicate with applications. (see Fig. 1.)

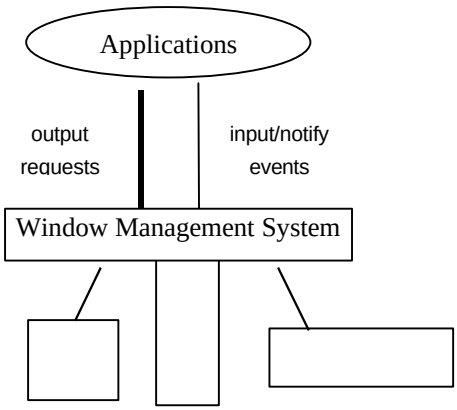


Fig. 1. The interactions between *Applications* and *Window Management System*.

The basic idea of shared window system is to intercept the events and/or requests flowing between WMS and applications, and distribute them among all participants.[] In some WMS (e.g. X-window system), the client/server model is adopted, and a standardized network protocol (e.g. X-protocol) is used to transmit those requests/events. We called them *network-aware* window systems. On the other hand, those *network-unaware* WMSs (e.g. MS Windows, Macintosh system) ignore potential telecooperation and result in high-degree interlocking among structures of applications, operation system, and I/O devices.[] The network-awareness will

significantly affect the implementation complexity of the requests/events intercepting.

2 Shared Window System Paradigms

As described above, a shared window system (SWS) is a mediate layer reside between applications and WMS to provide application sharing services. Hence, the capability to *redirect* request and/or event streams is essential. Dealing with the intercepted stream data in different ways, there are three sharing paradigms to build an SWS. The first one, called *event-sharing paradigm*, focuses on synchronizing of the events sent into/from shared applications. The second one, called *UI-image-sharing paradigm*, detects and refreshes the image content changing of the user interface (i.e. windows) of shared applications. The last one, called *request-sharing paradigm*, intercepts, distributes, and reproduces all output related requests and input events. In this section., we will discuss these three sharing paradigms in great detail.

2.1 Event-sharing Paradigm

The basic requirement of an event-sharing SWS is that all participants must execute the same application on their machines. By collecting and synchronizing input events produced, those applications will behave in the same way, such that the objective of share workspace can be achieved. From this point of view, the event-sharing SWS is pretty much like an *input synchronizer*. Fig.2. shows the system architecture of an event-sharing SWS. The implementation topics of event-sharing SWS include: *intercepting*, *collecting*, and *rerouting* of events. Therefore, an *event interceptor* module is invoked to catch events came from local/remote WMS; an *event reproducer* module is invoked to supply application the filtered events; and a *shared activity manager* module is invoked to collect all intercepted events and follow certain *floor control* rule to choose proper events for distributing.

Due to the shared applications must be run on all participant sites, the pure event-sharing SWS is naturally a *replicated* system. The filtered events is manipulated instantly by locally executed applications, so the event-sharing SWS can gain very short user responding time. Furthermore, the typical event packets are pretty small and consume very low network bandwidth to carry. However, a pure event-sharing SWS suffers from certain drawbacks. For example, the applications been shared must be *deterministic*. An application is deterministic if the output of application is determined by certain user input. Typical non-deterministic applications are those using random

number or local system status to display certain information on UI. Therefore, the event-sharing SWS could not guarantee, if non-deterministic applications are shared, same status can be reached by only distributing same input events. Moreover, the information been processed by event-sharing SWS contain only user input. Some advanced collaboration features (such as: latecomer joining, spontaneous application sharing, and cross-platform sharing) are difficult, even impossible, to implement.

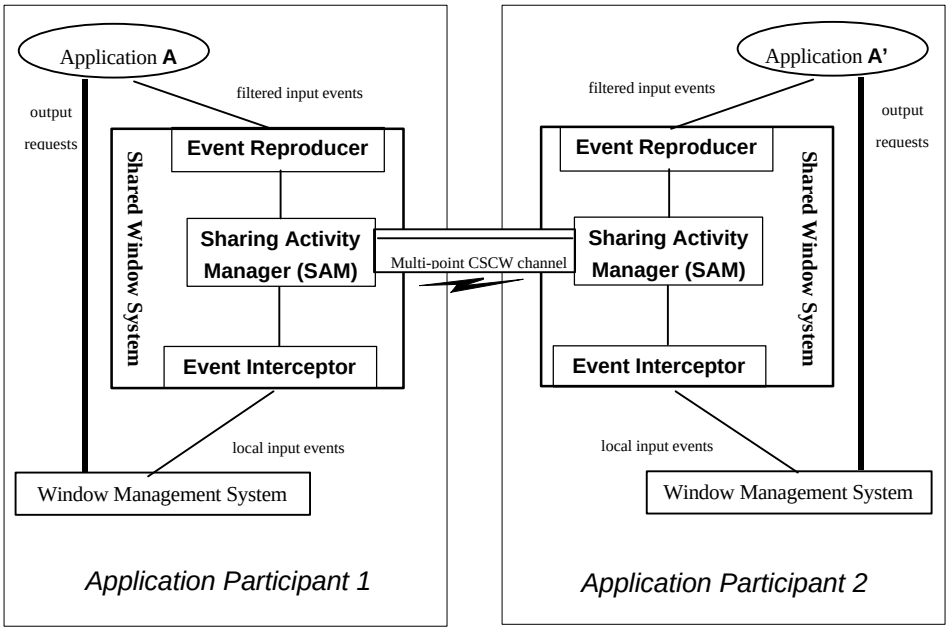


Fig.2. The *event-sharing* architecture.

2.2 UI-image-sharing Paradigm

Another intuitive thought to achieve shared workspace is to distribute the visual content of applications' user interface (UI) by *hard-copying* all UI images, we called it *UI-image sharing* paradigm. The shared applications can only be executed on *provider* site, and the other participants (called *sharer*) just receive the UI updating images sent from the provider and need not to run the same applications. Therefore, the UI-image-sharing SWS is naturally a *centralized* system.(see Fig.3.) The implementation criteria of UI-image-sharing SWS include: finding out of the proper occasion (e.g. when UI got updated) to capture UI images, reconstructing of the updating UI images, and minimization of the transmission bandwidth. (e.g. introduce a dirty-block detecting algorithm and/or image data compression algorithm) Moreover, input streams

collecting capability should be considered in order to provide multi-user concurrently operating ability.

Due to the UI updating of an application is achieved by sending output requests to underlying WMS, intercepting certain types of output request could be an adapted way to find out occasions to trigger UI grabbing function. The intercepted requests should be by-passed to WMS without any modification. In provider site, an *UI-image monitor* model is invoked to gather shared application's UI updating, and an *event reproducer* model is invoked to feed shared application the filtered input events. In sharer site, an *UI-image builder* model is invoked to reconstruct the original UI appearance of shared application. The *event interceptor* model is needed in both sites to collect all input events which will be further filtered by SAM obeying certain floor control rule.

Since the transmitted data is compose of input events and UI image blocks, it is possible for UI-image sharing SWS to support advanced collaboration features (such as: latecomer joining, spontaneous application sharing, and cross-platform sharing; see section 4). Furthermore, the centralized architecture of UI-image sharing SWS can eliminate the "deterministic shared application" problem which greatly harm the pure event-sharing SWS. The UI-image sharing SWS, however, needs far more bandwidth than an pure event-sharing SWS does.

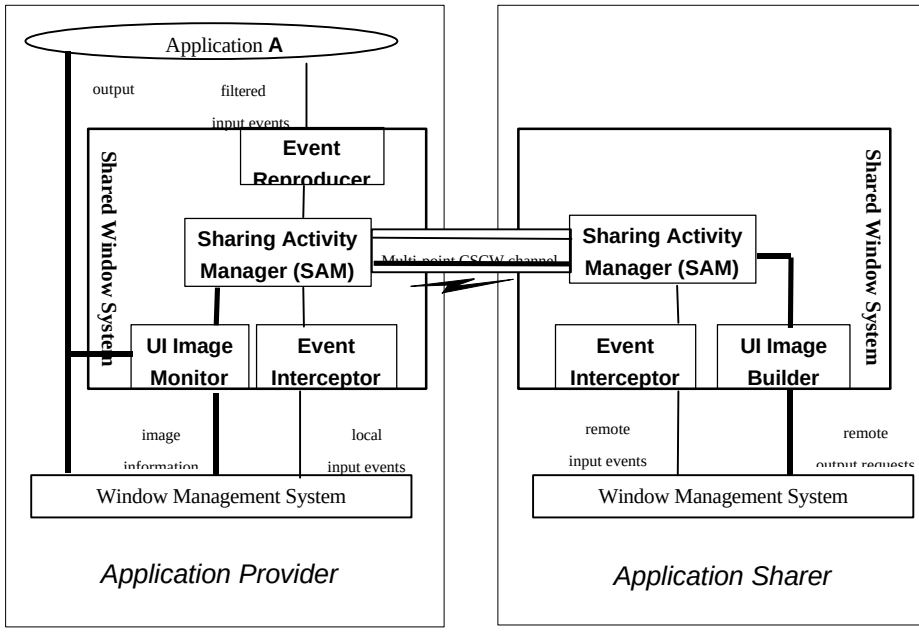


Fig.3. The *UI-image-sharing* architecture.

2.3 Request-sharing Paradigm

By comparison with distributing the visual content of applications, another way to achieve shared workspace among participants is directly multiplexing applications' *output requests*.[] We call this paradigm the *request-sharing* paradigm. The request-sharing SWS is also a centralized system. In other words, the shared application is only executed at provider site, and the SWS reside in sharer generates the "pseudo" application objects by reproducing those intercepted output requests. The system architecture of request-sharing SWS is shown as Fig. 4.

In provider site, the output requests of shared application are intercepted and analyzed by *request interceptor* model, and then, by-passed to local *request reproducer* model to accomplish original task. All request related local information (e.g. memory content pointed by pointers, data structure invoked resources, .etc.) must be further extracted and packed with request itself into *shared packets* which will be distributed by local SAM model to all sharers. In sharer site, the SAM model receives shared packets and unpacks them into original form, and then, feeds the *request reproducer* model those unpacked data for local reconstructing.

The accomplishment of output requests is highly independent to underlying WMS. For example, system object (e.g. window or resource) identifier (or handle) is only meaningful on local system and must be re-mapped while reconstructing them on other sites. In some WMSs, especially those network-unaware system like MS Windows, requests are tightly coupled with local operating system such that some requests use pointers to pass parameters. It could be a great challenge while implementing a request-sharing SWS on such WMSs.

However, the I/O requests of an application make WMS to generate real system objects, therefore, it is possible to accomplish certain UI updating (e.g. menu operating, window re-sizing,..etc.) by only using input events. The network bandwidth utilization can be more effective than that of UI-image sharing paradigm.

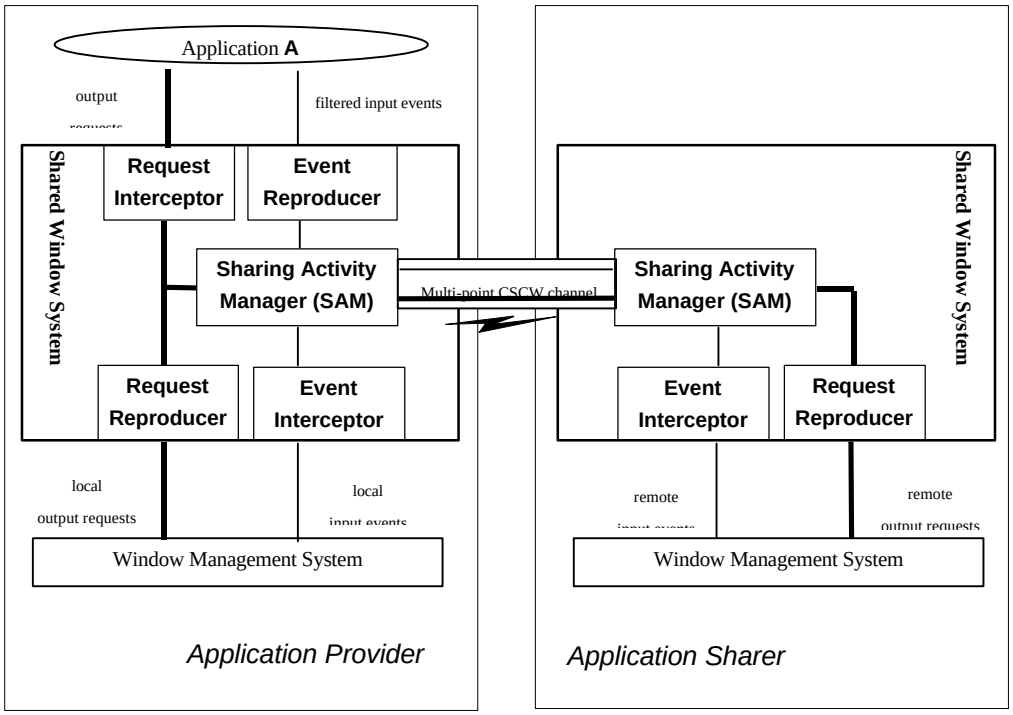


Fig.4. The *request-sharing* architecture.

Table 1. shows the differences among three sharing paradigms above. The discussion of lower three rows will be provided at section 4. We believe that the UI-image sharing paradigm is currently the best one for implementing of an SWS.

	Event-Sharing	**UI-image-Sharing**	**Request-Sharing**
Intercepted Data	input events	input events + UI- image blocks	input events + output requests
System Architecture	replicated	**centralized**	**centralized**
App. Deterministic	yes	**no**	**no**
Implementation Difficulty	easiest	**easy**	hard
Consumed Bandwidth	**lowest**	highest	medium
Response Time	**1-way network delay**	1-or 2-way network delay + UI-image constructing time	1-or 2-way network delay + requests constructing time
Cross-Platform Constrain (see Sec. 4)	yes	**no**	currently, yes
Latecomer Joining (see Sec. 4)	replay from beginning (log file needed)	**simple (log file unneeded)**	feasible (log file needed)
Spontaneous Sharing (see Sec. 4)	-	**simple (log file unneeded)**	feasible (log file unneeded)

Table 1. Comparison of three different sharing paradigms.

3 Generic Shared Window System Architecture

In recent times, the implementation of different sharing paradigm based SWS appeared in total different architectures.[][] The fact is that key functionality (such like: session control, floor control, transport control,. etc.) of an SWS is almost the same

among these different architectures, and need not to be re-designed from ground. A *generic sharing system architecture* which embodies replaceable different paradigms in single SWS is augmented to convince the inter-paradigm analyzing procedure. As Fig.5 shown, an SWS is separated into three components: *pseudo WMS layer* (PWL), *pseudo application layer* (PAL), and *sharing activity manager* (SAM). The PWL and PAL are sharing-paradigm-dependent and must be replaceable, while the SAM is sharing-paradigm-independent and can be reusable. Each component will be discussed in detail below.

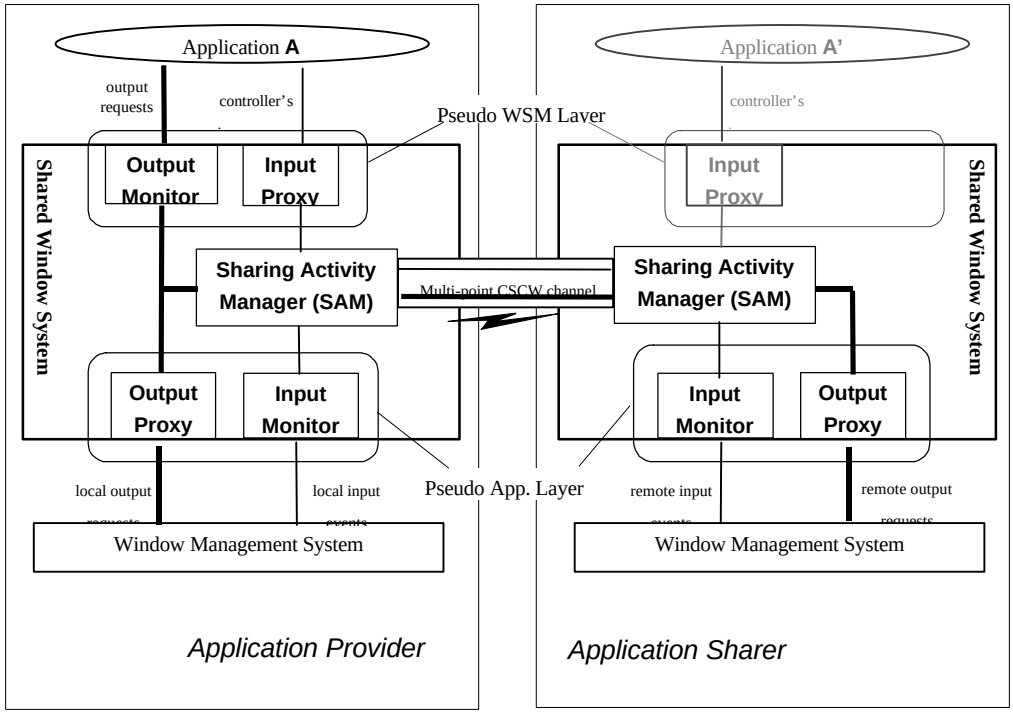


Fig.5. The *generic shared window* architecture.

3.1 Pseudo WMS Layer (PWL)

The aim of PWL component is to interact with shared applications such that applications can be shared in unaware. The role of PWL is just like a typical WMS which feed applications the proper events and/or accept output requests from applications. Two models, hence, is separated in further: *input proxy* which feed the filtered events, and *output monitor* which intercept and/or extract request related information. For *centralized system*, the shared applications are only executed on provider site, therefore, PWL is only performed on provider site.

In **event-sharing system**, the input proxy receives event packets from SAM,

translates them into the form that applications expect, and then feed them to applications. The output monitor is a dummy model and need not to do anything. All output requests are directly sent from applications to underlying WMS.

In **UI-image-sharing system**, the input proxy is the same as that in event-sharing system. The output monitor detects the UI updating of applications by monitoring certain output requests. It does not modify or even parse the content of requests but only obtain the opportunity for triggering image-grabbing process of output proxy (in PAL). All requests are by-passed (with the *trigger flag* which tell output proxy the UI content is changed) to output proxy which then sends them to WMS and/or perform the image-grabbing process.

In **request-sharing system**, the input proxy is also the same as that in event-sharing system. The output monitor intercepts all output requests of shared applications, extracts extra information (e.g. pointer data, resource data, .etc.) from applications if necessary, packs them into certain format, sends them to SAM, and then by-passes current request to output proxy (in PAL).

3.2 Pseudo Application Layer (PAL)

Form the view point of WMS, an SWS is just another typical application whose input and/or output behavior is determined by shared applications in certain degree. The aim of PAL component of SWS is to simulate the I/O behavior of shared application. There are two models in PAL to achieve this goal: *input monitor* which intercepts all events sent to applications by WMS, and/or *output monitor* which performs original output requests and/or grabs UI content updating. The PAL component is needed both in provider and sharer sites but accomplish different functions when playing different role.

In **event-sharing system**, the input monitor intercepts all input events which generated by users and sent out from WMS, packs them into certain format that SAM respect, then transmits to SAM for further filtering. The output proxy is a dummy model since we don’t pay any attention on output and totally rely on application themselves which running on all sites.

In **UI-image-sharing system**, the input monitor is the same as that event-sharing system. At provider site, the output proxy performs original requests from output monitor by simply by-passing to WMS. If *trigger flag* is set, output proxy must

execute the image-grabbing process to obtain, pack, and transmit the minimal dirty UI-image block to SAM. At sharer site, the output proxy receives and unpacks the dirty image block from SAM, then reconstructs the original UI-image by sending proper output requests to WMS. To obtain the minimal dirty UI-image block, it is helpful for output proxy to maintain certain UI related data structure, such as: top-most window list, window visible region, window content image cache, and window content dirty cache,. etc.

In **request-sharing system**, the input monitor is the same as that event-sharing system. At provider site, the output proxy simply by-passes current request from output monitor to WMS to perform original output task. At sharer site, the output proxy receives from SAM the request packets which contain extra request related data. Those extra data must first be restored on local system for further pointer or ID referring. Moreover, all resource IDs (or handles) should be translated into local value to match the local system status of sharer site. It is necessary to maintain an ID mapping table for this purpose. When all parameters of current request are properly processed (i.e. restored and re-mapped), the output proxy could safely reproduce that request.

3.3 Sharing Activity Manager (SAM)

The SAM component is the common and paradigm-independent part of any SWS in our architecture. It centrally manages all application sharing activities on local site and works together across network with other SAMs to exchange information. An typical SAM provides following capabilities:

- **Basic conferencing related functionality**, includes: sharing session management (e.g. open, join, leave, and close), and packet exchanging. In order to hide the low-level network communication detail and to accomplish the network interoperability, adopting a standardized conferencing protocol (i.e. **T.120** serial specifications) could greatly simplify the implementation of SAM.
- **Floor control mechanism**, by which the SWS determine which participant gain the control right of the shared application. Several fundamental floor control functions such as: requesting, confirming, releasing, passing, and taking of the floor, should be provided within SAM component.[]
- **SWS specific capabilities**, for example: *telepen* (or annotation) which provide

freehand drawing capability within shared workspace, and *telepointer* which provide visual cue for multiple user positioning.

Besides, there are some optional capabilities which can improve the sharing quality, such like: virtual desktop, palette (or color table) optimization, nearest font face matching, .etc. All of them are related to default output capability (or system property) of local WMS, we called it *WMS capability localization problem*. Since the capability localization problem is sharing paradigm independent, it is better to be handled by SAM. This problem could be solved by *capability negotiating*.[T.SHARE]

4 Issues and Solutions

To be an useful collaboration tool, the SWS should meet some human-human interaction requirements, for example: *latecomer joining* which allow non-initial participants to join sharing session, *spontaneous application sharing* which allow an already executed application to be shared, and *cross-platform sharing* which allow sharing session be arisen among different WMS platforms. All these issues and corresponding solutions could be paradigm-dependent.

4.1 Latecomer Problem and Solution

When the latecomer joined into a sharing session, the sharing activity had already proceeded for a while and, therefore, the shared applications were no longer at the initial state. To make sure the SWS of the latecomer could accept the subsequent sharing packets, the state of shared applications on the latecomer site must be synchronized by some ways to that of other participant sites.

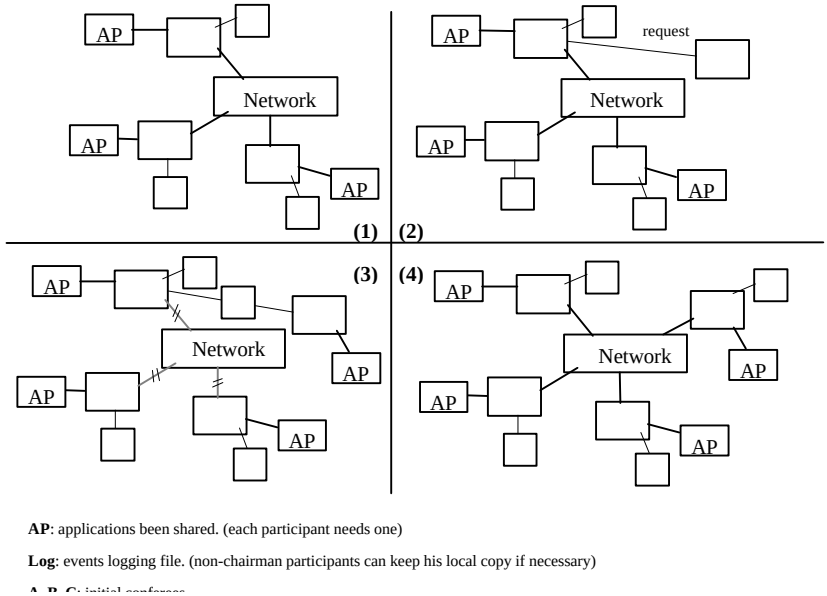


Fig.6-(1). Latecomer joining of *event-sharing* paradigm.

For **event-sharing** paradigm, the information be processed by SWS contains only input events. To make shared applications, which was initiated on the latecomer site, to reach synchronous state, the only way is through the *replaying* of all events generated within sharing session. An *event log file*, in which all events were recorded, is needed to accommodate event replaying. The log file could be kept within all initial participants for the reason that the latecomer could receive it from nearest participant. Fig.6-(1) illustrates the four stages of latecomer joining of event-sharing paradigm.

The participant A is the nearest participant to latecomer L, and is assigned to be the log file supplier while L asking to join. Before L accomplishing his joining, the whole sharing session should be halted to prevent potential problems (e.g. packet lost, out of synchronization, ..etc.). As long as the log file was transmitted and replayed without any lost, the latecomer joining could be completed. However, the resulting state of shared application of L is not guaranteed to be the same with that of A if the shared application is *non-deterministic*. Moreover, the time spent to accomplish L's joining is increased as the session progressing.

For **UI-image-sharing** paradigm, the current state of the shared application could be entirely determined by the UI appearance. While the latecomer requests for joining, it is sufficient to synchronize the UI status by simply grabbing certain window images on provider site and reconstructing them on latecomer site. To efficiently

accomplish image grabbing and reconstructing tasks, it is essential for application provider to maintain a top-most window list (or hierarchy) of the shared application. Fig.6-(2) illustrates the four stages of latecomer joining of UI-image-sharing paradigm.

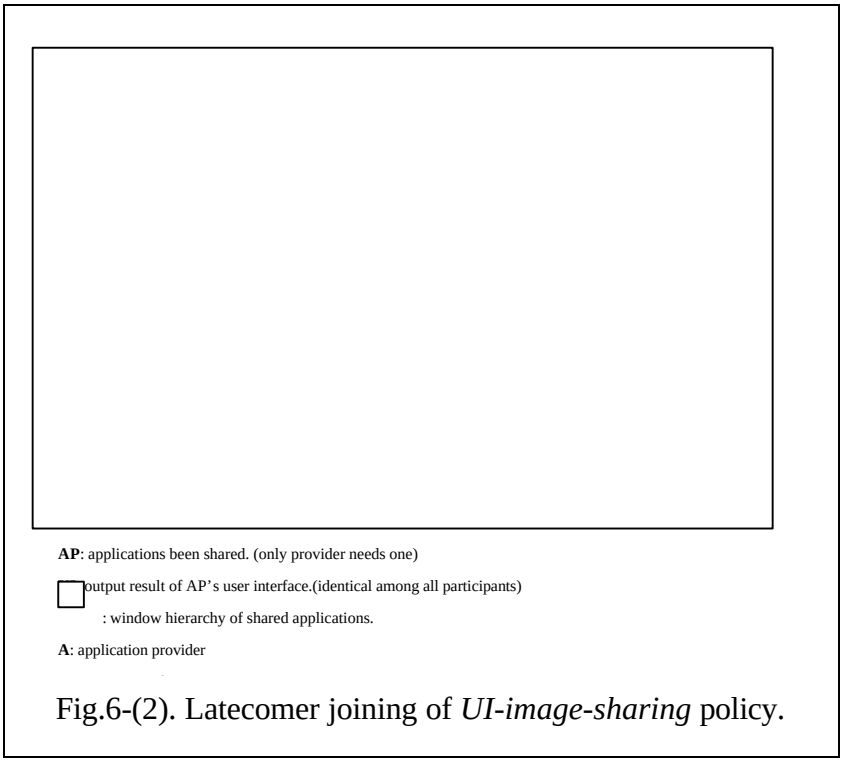


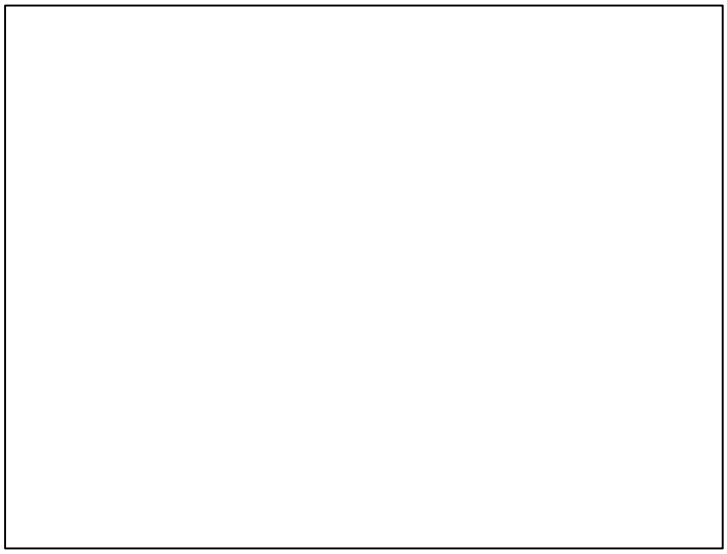
Fig.6-(2). Latecomer joining of *UI-image-sharing* policy.

Participant A is the provider of the shared application. When A received the joining requests from latecomer L, it halted the sharing session, extracted the window hierarchy of shared application, captured the content images, and then transmitted the window hierarchy and images to L. After the arrival of first packet, L started to reconstruct UI contents according to the received information. A could continue the sharing session with L after receiving the L's complete-joining acknowledgment.

No matter how late L come, the time required to join an UI-image based sharing session is almost the same and is dependent on the number and the size of window involved.

For **request-sharing** paradigm, in fact, the shared application was reproduced on sharer site by creating real system objects. There were tow approaches to achieve latecomer joining. In first approach, *all* requests and events are recorded by provider and reproduced by latecomer. It is easy to implement but suffered from the drawback of required time and space increasing while session proceeding. In second approach, which was proposed and implemented on X-Window by Chung [Chung 93], an object

status table is maintained by provider and transmitted to latecomer for reconstructing. It can eliminate the obstacle of time/space increasing of later joining. Fig.6-(3) illustrates the four stages of latecomer joining of request-sharing paradigm using Chung's approach.



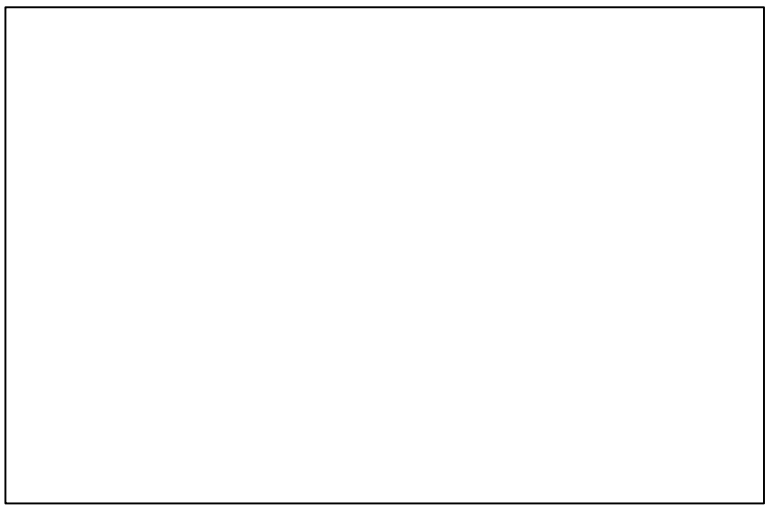
AP: applications been shared. (only provider needs one)
UI: output result of AP's user interface.(identical among all participants)
: internal data structure of the shared window system. (such like: resource/window hierarchy)
A: application provider
B, C: initial conferees.
L: latecomer conferee.

Fig.6-(3). Latecomer joining of *request-sharing* policy.

4.2 Spontaneous Application Sharing and Solution

» © ¼ ¤ À ¥ ¦ ¸ © Ê @ É Ð ® ² § P § @

- Â I



AP: applications been spontaneously shared. (only provider needs one)

output result of AP's user interface.(identical among all participants)

: window hierarchy of shared applications.

Fig.7-(1). Spontaneous sharing of *UI-image-sharing* architecture.

AP: applications been spontaneously shared. (only provider needs one)

output result of AP's user interface.(identical among all participants)

: internal data structure of the shared window system. (such like: resource/window hierarchy)

Fig.7-(2). Spontaneous sharing of *request-sharing* architecture.

The second method is hard to implement since that all WMSs do not support comprehensive status querying/restoring services of run-time system objects.

4.3 Cross-platform Sharing Problem and Solution

» © ú ³ ¤ Î @ É [º ¤ Ú Á¹ F ì ¾ ¥ x @ É ¥ ï ® Ê

- packet translation

- general transport protocol

5 Recorder and Player

» ©¡ ¡ú b¦ @¨ É¡ µ¤ § Þ N¦ b° Q¿ ý ¼¶ Ø¤ ñ Wª À ¥ Î

6 Conclusion

In this paper, we propose a generic shared window architecture which regulate the implementation concept and hide the adopted sharing paradigm. Since the most important sharing management functions are independent to the adopted sharing paradigm, it is possible to reuse same sharing management module among different sharing paradigms based SWS. In this generic architecture, an SWS is consisted of three system components: pseudo application layer (PAL), pseudo WMS layer (PWL), and sharing activity manager (SAM). The first two components (PAL and PWL) are paradigm dependent and must be replaced from different paradigm based SWSs. The third component (SAM), which appeared as the kernel of an SWS, is paradigm independent and could be reused.

- ´ ¥ X¤ £ P¦ @¨ É¡ ² ¤ U¤ Ä d° I- ¤ s° Ã D¦ A¿ ð ì ì Ý Ã D¦ À ¥ µ¡ {¦ ¦ Û © Ê @¨ É Ã D³ ¸ Ñ M¤ è k
- ´ ¥ X¹ F¦ ¨ ¥ ¥ x µ ¦ @¨ É º Å é ® ×
- ´ ¥ X¹ F¦ ¨ ¥ ¥ x µ ¦ @¨ É º Å é ® ×

7 Reference