Synchronous IPC over Transparent Monitors

Trent Jaeger Jonathon E. Tidswell * Alain Gefflaut
Yoonho Park Kevin J. Elphinstone † Jochen Liedtke †
IBM T.J Watson Research Center
Hawthorne, NY 10532
Email: sawmill@us.ibm.com

1 Introduction

Interprocess (IPC) monitoring enables the examination of any IPC between a *source* and a *destination*. IPC monitoring is useful for a variety of purposes, including debugging, logging, and security. For example, a monitor may collect communication state for the purpose of debugging a program consisting of several independent tasks. Also, a monitor can be used to filter communication data or control the communication rate for security purposes.

Transparent monitoring means that the source and destination are not aware that they are being monitored. This has two advantages: (1) the system can control the insertion and removal of monitors without interacting with either the source or destination and (2) the source and destination protocols do not need to take into account the possibility that they may be monitored. Traditional systems make no attempt to support transparent monitoring. Using Mach-style *ports* [1], the source and destination hold rights that must be revoked in order to insert a monitor, so at least one must be notified before such a change can occur safely. The Pebble microkernel enables transparent redirection of source IPCs using its *portals* to implement customized IPC, but the redirection is not transparent to the destination because it sees that the message is from the redirected task, not the original source [3].

Other IPC mechanisms, such as Clans & Chiefs [6] and IPC Redirection [2], enable monitors to intercept and forward IPCs while claiming to be the original source of the IPC. Thus, the destination receives the IPC from the source, not the monitor, so it need not know that an IPC is being monitored. Unfortunately, such mechanisms are not truly transparent because the kernel's IPC semantics are not preserved when a monitor intercepts an IPC. Modern microkernels implement a synchronous IPC semantics, which means that the source is blocked until the destination is ready to receive the IPC or an error occurs (e.g., the destination task is killed or a timeout expires). When destination commences receipt, the IPC is sent to the destination and the source unblocks. Unfortunately, if a monitor is inserted on an IPC path, the source is unblocked when the IPC is received by the monitor, not the destination. This may result in some anomalous behaviors, such as: (1) the source assuming that IPCs have been delivered to the destination before they really have; (2) the source terminating IPCs due to timeout expiration even though the destination is ready, but because the monitor was not ready; and (3) the source assuming that the IPC was delivered reliably to the destination when an error may have occurred.

In this paper, we propose an IPC mechanism that restores synchronous IPC semantics over transparent monitors. The key feature of this mechanism is that system monitors are considered as an extension of the kernel, so the source and destination are treated as if the kernel is still processing the IPC. However, there are a number of possible monitoring extensions that must be considered, and these introduce a number of problems that must be solved simultaneously. Our design enables layering of these additional semantics upon system monitors as necessary.

The remainder of the paper is as follows. In Section 2, we define the basic synchronous IPC semantics that must be achieved and refine these semantics for relevant extensions. In Section 3, we construct a synchronous IPC protocol that solves these problems by incrementally extending a basic IPC mechanism. In Section 4, we demonstrate the use of the synchronous IPC mechanism on an example which both redirects and controls the rate of IPCs securely. In Section 5, we examine efficient implementations of the synchronous IPC mechanism upon the L4 microkernel using the IPC redirection mechanism. In Section 6, we conclude the paper.

*School of Computer Science and Engineering, University of New South Wales, Australia
†Faculty of Informatics, Universität Karlsruhe, Germany

2 Synchronous IPC Semantics

We now define precise synchronous IPC semantics. In general, synchronous IPC means that the source blocks until the destination receives the IPC at which time the source is unblocked. However, there are three further requirements that complicate the definition of these semantics: (1) the monitors may change the identity of the source or destination; (2) monitors may wish to hide the timing of the destination's receipt of an IPC from the source; and (3) the monitors themselves may be of different trust levels. Additionally, a source may desire an unreliable IPC (i.e., be unblocked with no guarantee of delivery). This is easily handled by enabling the source to signal that synchrony is not required, so we do not address it again until the implementation (see Section 5).

2.1 Basic Synchronous IPC

Basic synchronous IPC has the following semantics:

- **Reliability**: The source blocks until the IPC is delivered to the destination or an error occurs.
- **Error Handling**: The source receives an error if the IPC is not delivered to a destination. The destination also receives an error if it is not able to properly receive an IPC.
- **Timeout Expiration**: An IPC is terminated if a destination takes longer than a source-specified timeout to commence receiving an IPC. Similarly, an IPC is also terminated if a source takes longer than a destination-specified timeout to commence sending an IPC.

First, synchronous IPC is reliable, such that an IPC is not complete until it is received by a destination. Thus, the source is blocked until it is known that the IPC was received by the destination. In current systems, the source is unblocked when the IPC is received by the monitor, not the destination, so communication is unreliable by default.

Second, synchronous IPC semantics mean that an error is signalled if the IPC is not delivered to a destination. The source is always signalled. The destination is signalled if it is waiting for an IPC precisely from this source. It must be possible to express a variety of error codes to the source and/or destination. In current systems, once an IPC is delivered to the monitor, the source will not receive a delivery error message.

Third, both the source and destination may set timeouts. These timeouts indicate the amount of time that the source is willing to wait for the destination to commence receipt of an IPC or for a source to commence sending an IPC, respectively. Unfortunately,

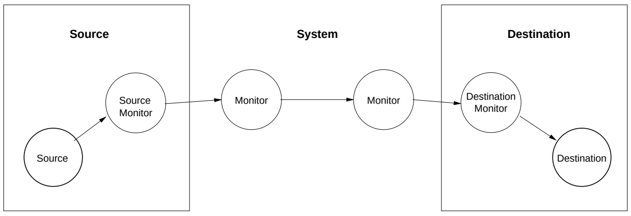

Figure 1: Monitors may be inserted by the source, destination, or system. Source and destination monitors are adjacent to the those tasks are viewed as extensions of the source and destination by the system.

a monitor may disrupt the interpretation of timeouts. For example, if a monitor does not have a thread ready to receive an IPC being sent with a zero timeout, then a timeout error may occur even though the destination is ready to receive the IPC. Such an error violates the transparency of the monitors because the source and destination must recover from an error caused by a monitor.

2.2 Changing Source/Destination

In the Clans & Chiefs [6] and IPC redirection mechanisms [2], it is possible, within some restrictions, for a monitor to redirect an IPC to another destination or change the identity of the source from which the destination receives the IPC. Regardless of these changes, synchronous IPC is defined with respect to the original source. Therefore, the original source will remain blocked until the IPC is delivered to a destination (i.e., can be sent to the destination specified in the IPC) or an error occurs.

2.3 Controlled Synchrony

By default, the synchronous IPC semantics specifies that the source is unblocked when the IPC is delivered to the destination. We identify two cases where this semantics is unacceptable: (1) to prevent covert channel creation, a monitor may not want to the source to be able to detect the time it takes a destination to receive an IPC and (2) a monitor may generate multiple IPCs from a single IPC send, so it may want the source to be blocked until all the IPCs are sent. In the first case, a trusted task, such as the Naval Research Lab (NRL) Pump [4], is placed between the source and destination. This task normalizes the delay for the source to receive confirmation that an IPC has been delivered reliably to mask the processing behavior of the destination. Thus, it must be possible for a monitor to be able to control when the source is unblocked.

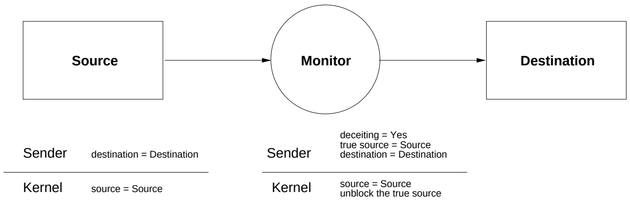

Figure 2: **Basic Synchronous IPC Mechanism**: The source is blocked when it sends the IPC and unblocked when the IPC is received by the destination. This identity of the *blocked source* is maintained in the *true source* of the IPC. The *sender* parameters indicate the inputs supplied by the sender of the IPC, and the *kernel* parameters indicate the additional inputs supplied by the kernel on the IPC.

2.4 Untrusted Monitors

As shown in Figure 1, monitors may be loaded by the system, the source, or the destination. Monitors loaded by the source or destination may not be trusted by the system, so it may not be possible to depend on these monitors to implement synchronous IPC. For example, a destination monitor may leave a source blocked forever without either delivering the IPC or returning an error. Since we can assume that these monitors are under the control of the source or destination, respectively, we can provide synchronous IPC between the source and destination domains. It is up to the source and destination monitors to unblock the original source and ensure that the IPC is delivered properly, respectively.

3 Synchronous IPC Mechanism

The key assumption in the synchronous IPC mechanism design is that *the system monitors are assumed to be trusted extensions of the kernel with respect to maintaining IPC state.* For any security-sensitive monitor, such an assumption is quite reasonable because the security of the system is dependent on the correctness of this monitor. As described above, the source or destination may insert a monitor that they trust, but a monitor must be trusted by the source and destination to particpate in the synchronous IPC mechanism.

3.1 The Basic Mechanism

The basic mechanism is shown in Figure 2. First, when a source sends an IPC that is redirected to a monitor, the source is blocked on the IPC's destination. Next, the monitor receives the IPC, does its processing, and forwards the IPC to the destination. Since IPCs from this source to this destination are redirected by the kernel to this monitor, the monitor can send an IPC to this destination which it claims to be from this source (IPC deceit [6]). In addition to the traditional IPC information (e.g., destination and message), the *true source* field of an IPC is added; if deceit is authorized, this field specifies the source of an IPC. Finally, the IPC is received by the final destination. At which time, the kernel unblocks the source and, for a deceited IPC, the true source of the IPC.

3.2 Changing Sources

A monitor may claim that an IPC is from another task if the monitor can deceit for that task as well. To do this, the monitor specifies this other task as the true source of the IPC. However, the kernel needs to know the original source in order to unblock it when the IPC is delivered to the destination. Therefore, the IPC information is extended by adding a field for the *blocked source*. The basic mechanism is extended in the following way: when an IPC is received by the final destination, the kernel unblocks the source (i.e., the monitor or a source which can send directly). If the IPC is deceited and a blocked source is specified, then the kernel unblocks the blocked source. If the IPC is deceited and no blocked source is specified, then the kernel unblocks the true source.

3.3 Changing Destinations

In addition, a monitor may redirect an IPC to a completely different destination. Since a thread can only send an IPC to one destination at a time, it is not necessary to maintain the "blocked destination" for a blocked source. The monitors must include the proper true/blocked sources in order to ensure that the IPC appears to be from the correct source and the actual blocked source is unblocked when the IPC is delivered to the new destination.

3.4 Error Handling

This synchronous IPC mechanism must detect and be able to report errors to the blocked source. First, if a monitor sends an IPC in which the kernel detects an error (e.g., destination is not a running task), then the monitor is notified that there was an error in the attempted send. The monitor may have an error handling strategy, but typically, it will simply return the error to the source. Since the source is blocked on a send, the monitor must be able to get the kernel to signal an error to the source. To handle this, the monitor sends an *error IPC* to the blocked source[1]. The error code can be encoded in the IPC message in a standardized format.

[1] We use the term blocked source to refer to the thread that is blocked on the IPC regardless of whether it is specified as the true or blocked source in the IPC.

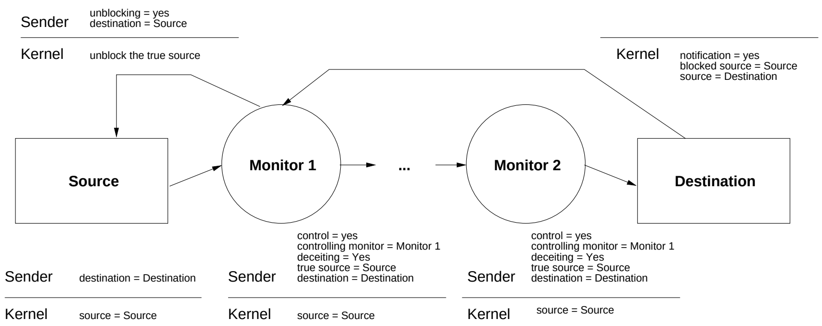

Figure 3: **Controlled Synchrony**: If a monitor wants to prevent the blocked source from being unblocked when the IPC is delivered, it sets itself as the *controlling monitor* for the IPC. When the IPC is delivered to the destination, the kernel generates an IPC to the controlling monitor rather than unblocking the blocked source. The *sender* parameters indicate the inputs supplied by the sender of the IPC, and the *kernel* parameters indicate the additional inputs supplied by the kernel on the IPC.

3.5 Interpreting Timeouts

When an IPC is sent, the source may set a timeout limiting the duration for which it will wait for the destination to commence receipt of the IPC, called a *send timeout.* Some kernels, such as L4, also permit the use of untrusted pagers, so for these systems a second timeout signifying a limit on the duration for delivering the IPC may be set, called a *delay timeout.* Conversely, the destination may set corresponding receive timeouts for the source. Since the receive timeouts are the directly opposite, we only describe source timeouts here. Also, other timeouts, such as real-time, may be set, but these are verified similarly to the send timeout.

In general, any source IPC timeouts should be checked against the ultimate destination, not the monitors. In addition, the timing behavior of the destination must be hidden from the source to prevent creation of a covert channel. Therefore, assuming monitors receive redirected IPCs in a reasonable amount of time, which should be a reasonable assumption for trusted monitors, timeouts have the following semantics. First, since the system monitors are considered to be part of the kernel logically, the sender's send timeouts are not checked until the IPC is delivered to the ultimate destination. This enables the monitors to catch any timeout error rather than the source, and enables the monitors to change the destination without invalidating timeout checks. The destination's send timeouts are relative to the state of the source, so the fact that the source is blocked on a send to the destination can be verified by the kernel. Since the source can send to the destination no covert channel problems occur. If the destination upon which the block source is blocked upon is changed by a monitor, destination send timeouts must be re-evaluated. In this case, the new destination's send timeout must be checked, and the old destination's send timeout becomes active again (i.e., is set based on the original receive time).

Second, delay timeouts are checked only when the message is actually delivered to the destination. This again is consistent with the original semantics. However, the delay timeout specified by the source may be significantly larger than the delay timeout desired by the monitor. Therefore, the lesser of the two delay timeout values is used.

3.6 Controlling Synchrony

In order to control when the synchronous IPC is completed, a monitor must be able to control when the source unblocks. The synchronous IPC mechanism is extended to enable this control as shown in Figure 3. There are three new steps in the synchronous IPC mechanism: (1) setting the controlling monitor; (2) notifying the controlling monitor; and (3) unblocking the blocked source.

First, when a monitor wishes to control when a blocked source is unblocked, it assigns itself as the *controlling monitor* of the IPC. It does this by adding an entry in the IPC including itself as the controlling monitor when it forwards the IPC to the next monitor or destination. Only one monitor can be the controlling monitor at a time although a sequence of controlling monitors can be constructed as described in the subsequent section. Since all system monitors are trusted to perform the algorithm correctly and the system monitors follow the source monitors, then any system monitor can choose to become the controlling monitor.

Second, when the IPC is delivered to the destination or an error is signalled, the kernel generates an *controller notification IPC* for the controlling monitor. Such an IPC includes the identity of the blocked source and the actual destination, so the monitor can determine what action to take. The identity of the actual destination may be useful if the action that the monitor takes also depends on it (e.g., an NRL pump for multiple high targets).

Third, the monitor unblocks the source by sending an unblocking IPC to the source. An unblocking IPC is indicated when the monitor sets the unblocking bit in an IPC to the source.

3.7 Creating Controller Chains

If a second monitor sets itself as the controlling monitor on the same IPC path, a chain of controlling monitors is created. When a monitor sets itself to be the controlling monitor of an IPC and another monitor has already set itself to be the controlling monitor, then the latter monitor must record the association between the preceding controlling monitor and the IPC path (i.e., blocked source). When the IPC is delivered to the destination, the last controlling monitor in the chain is notified as described above. This monitor then retrieves the preceding controlling monitor for this block source, if any, and triggers a controller notification IPC for the preceding monitor. This is repeated until the blocked source can be unblocked.

3.8 Untrusted Monitors

We assume a trust model in which only the source trusts its monitors and only the destination trusts its monitors. System monitors are trusted by all. In this case, we define IPC to be synchronous between the last source monitor (including the source) and the first destination monitor (including the destination) on the IPC path.

When a source sends an IPC, it trusts its monitors to deliver that IPC to the system monitors with the appropriate values for the true source, blocked source, and controlling monitor. The system monitors can change the true source and controlling monitor as necessary and are trusted to unblock the blocked source.

When the IPC is delivered to the first destination monitor, this is treated the same as if the IPC was being delivered to the destination itself. Since the destination installed these monitors, this functionality could just as easily be part of the destination. Therefore, all actions taken when the IPC is delivered to the destination are taken now: (1) the delay timeouts are checked; (2) the controlling monitor, if any, is notified; and (3) otherwise, the blocked source, if any, or the true source is unblocked.

4 Example

Consider a system depicted in Figure 4 with two security levels, public and secret, where a public client may be logging information on a secret file system. The secret file system must not leak any information to the public client which is logging information in it, but the secret file system is not trusted to prevent such a leak (e.g., it may contain a Trojan horse). Although the secret file system cannot send an IPC to the public client, it could leak information through a covert channel. Therefore, a NRL pump-style monitor (see Section 2.3) is installed between the public client and the secret file system to normalize the timing behavior of the secret file system. This monitor sets the true source to the public client and sets itself to be the controlling monitor. When the IPC is delivered to the secret file system, the pump is notified and, eventually, unblocks the source as described in Section 3.6.

Next, the secret file system is replaced by a new secret file system task. Another monitor is added to redirect the public client's log IPCs to the new system task transparently. Since the new secret file system also may contain a Trojan horse, the NRL pump is left on this IPC path. In this case, the destination of the IPC is changed by the redirecting monitor, but the true source and controlling monitor remain the same. Thus, the NRL pump is notified when the IPC is delivered to the new secret file system. The notification message contains the identity of the blocked source, so the correct source is unblocked.

Lastly, the client may add its own monitor in hopes of detecting some information about the behavior of the secret file system. This monitor sets itself to be the controlling monitor. When the IPC is sent from the public client, it is immediately delivered to the public client's monitor, so no knowledge about the behavior of the secret file system is obtained. The NRL pump sets itself to be the controlling monitor, so it is notified when the IPC is delivered. Therefore, it can control when the public client's monitor is notified about the IPC delivery.

5 Implementation Issues

We consider implementation of the synchronous IPC mechanism presented above on the L4 microkernel [5] using the IPC redirection mechanism [3]. This implementation must enable representation and application of the IPC state. The IPC state is divided into two types: (1) IPC options and (2) IPC parameters. First, there are eight meaningful IPC options as shown in Table 1, so three control bits are needed to represent them. Second, the current L4 microkernel implementation supports a parameter for deceiting already (i.e., true source), so only the blocked source and controlling monitor parameters must be maintained additionally.

The IPC options in L4 are passed in the *send descriptor* and received in the *message dope*. Currently, one bit is devoted to such information in the send descriptor, so two additional bits are necessary. The direct effect of this is that the send message buffer must be 16 byte-aligned (rather than 4 byte-aligned). The message dope already has three control bits, so no new information is necessary.

There are three choices for maintaining the blocked

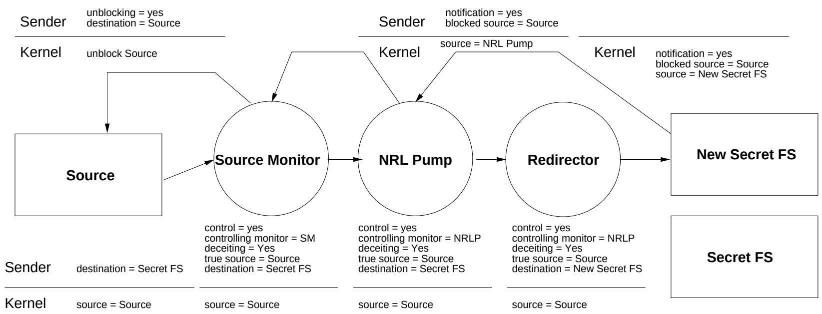

Figure 4: **Monitoring Example**: The NRL pump and redirector monitors are system monitors that control the synchrony between the public client (and its monitor) and the secret file system and redirect the IPC to a new secret file system, respectively. The *sender* parameters indicate the inputs supplied by the sender of the IPC, and the *kernel* parameters indicate the additional inputs supplied by the kernel on the IPC.

Option	*Action*
0	unreliable send
1	error
2	notification
3	unblocking
4	deceit
5	deceit w/ blocking source
6	deceit w/ controlling monitor
7	options 5 and 6

Table 1: IPC options for the complete synchronous IPC mechanism.

source and controlling monitor: (1) pass as IPC parameters; (2) store using kernel system calls; and (3) encode it via the monitors. In the first case, the additional IPC parameters are included in the IPC system call, but only where necessary. In the second case, system calls are used to store the additional IPC parameters in kernel data structures. In the third case, monitors encode the IPC parameters using other parameters. For example, a monitor can encode the fact that it is a controlling monitor by setting itself to be the blocked source.

We prefer the first choice because it is simple and efficient in the normal case. The kernel already handles the IPC state in parameters, so this choice adds little new complexity to the kernel or monitors. Using the second and third choices would require new system calls and data structures for the kernel and/or monitors which we hope to avoid. From a performance standpoint, the first option will typically be no different than the current L4 IPC mechanism. We expect that changes in the source will be rare, and the current examples that use the controlling monitor are not performance critical, and, in fact, purposely slow performance.

6 Conclusions

Transparent monitoring enables the dynamic insertion of monitoring functionality without requiring either the cooperation or the awareness of either of the tasks being monitored. Therefore, it is preferable to traditional overt monitoring mechanisms. However, in order to implement transparent monitoring, the kernel's IPC semantics must be preserved regardless of how many monitors exist and what actions they may take. In this paper, we presented a synchronous IPC mechanism that is preserves IPC semantics over transparent monitors. This mechanism considers system monitors to be an extension of the kernel, so the source can be blocked until all the monitors have completed their actions and forwarded the IPC to the destination. Therefore, synchronous IPC is possible even given a variety of monitor actions, such as redirecting IPCs to a new source, changing the source of an IPC, and changing the communication rate. Also, this mechanism supports useful variations of synchronous IPC semantics, such as requesting unreliable communication, delaying delivery notification, and using untrusted monitors.

References

[1] R. V. Baron *et al.* Mach kernel interface manual, 1990. Unpublished manuscript from the School of Computer Science, Carnegie Mellon University.

[2] T. Jaeger *et al.* Flexible access control using IPC redirection. In *Proc. 7th HotOS*, 1999.

[3] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble component-based operating system. In *USENIX Technical Conference*. USENIX, June 1999.

[4] M. H. Kang and I. S. Moskowitz. A pump for rapid, reliable, secure communication. In *Proc. 1st CCS*, pages 119–129, 1993.

[5] J. Liedtke. Lava Nucleus (LN) Reference Manual: 486, Pentium, Pentium Pro, version 2.2. Available at http://i30www.ira.uka.de.

[6] J. Liedtke. Clans & chiefs. In *Architektur von Rechensystemen*. Springer-Verlag, March 1992. In English.