The b^2/c^3 problem: How big buffers overcome covert channel cynicism in trusted database systems

J. McDermott

Naval Research Laboratory , Code 5542, W ashington, DC 23075, USA

Abstract

We present a mechanism for communication from low to high security classes that allows partial acknowledgments and flow control without introducing covert channels. By restricting our mechanism to the problem of maintaining mutual consistency in replicated architecture database systems, we overcome the negative general results in this problem area. A queueing theory model shows that big buffers can be practical mechanisms for real database systems.

Introduction

Kang and Moskowitz [8] presented a general mechanism for rapid and reliable communication from low to high security classes. The mechanism, called the *Pump*, includes an adjustable and easily quantifiable covert channel to provide acknowledgments. Their general result is that reliability, performance, and security cannot be achieved together. This negative result agrees with other work [10] and we do not dispute it here. Instead, we present positive results for a useful special case of communication from low to high classes: maintenance of mutual consistency in replicated architecture multilevel-secure database systems.

The replicated architecture [6] is an approach to providing strong multilevel security in database systems. It provides multilevel security by replicating single-level copies of low sensitivity data into higher classes. The replicated architecture depends upon the ability to write-up reliably without creating an undesirable information flow. According to the Bell-LaPadula model [1], write-up without read access is permissible. This kind of write-up is performed to volatile storage, without acknowledgment. Furthermore, it requires the use of memory descriptors and mechanisms that do not carry read access permission to the destination memory segment, a feature rarely supported by existing hardware. The latter problem can be overcome by simulating the write-up with a read-down, but the lack of coordination and the volatile nature of the destination memory segment remain problematic.

By exploiting the structure of a computation, Sandhu, Thomas, and Jajodia [13, 14] have shown how write-up without acknowledgment can be used in object-oriented systems. Kang and Moskowitz have proposed a general mechanism for writing up reliably with recovery by using acknowledgment with a controlled bandwidth [1] covert

1. Moskowitz and Kang [12] argue that the concept of bandwidth is not a sufficiently precise measure of the vulnerability introduced by a covert channel and provide a new metric, the *small message criterion (SMC)*. The small message criterion depends on a triple (n, τ, ρ): when a covert channel exists in a system, the SMC gives guidance for what will be tolerated in terms of covertly leaking a short message (e.g. master key) of length n bits in time τ with fidelity of transmission ρ%.

channel. Kang and Moskowitz assert that, for the general case, one cannot have write-up that is reliable, recoverable and secure. The thesis here is that, for an important special case, this is not so. Our special case is write-up performed for the purpose of maintaining mutual consistency in the replicated architecture.

Three advantages of restricting our solution to the replicated architecture are: 1) bounded storage space requirements at the destination, 2) a relatively small number of source and destination processes, 3) transaction management. Because we are only writing up for the purpose of replicating data items in a database, we know that no new objects are created by writing up to higher classes[1]. We can fix the total storage available at lower security classes and thus bound the total replicated storage for all higher security classes. Because we are only supporting database system instances, we know there will not be a large number of readers and writers[2]. Because we are only supporting systems with transaction management capability, we can choose to discard some write-ups in a correct fashion, in the event of a failure, and bring the replicas into convergence with later transactions. This latter point is proved by Bernstein, Hadzilacos, and Goodman [2].

Our specific problem is to provide a service for propagating update projections in the replicated architecture database system. This service is to be *reliable, recoverable*, and *secure*. By secure we mean free from implementation invariant covert channels and compliant with a Bell-LaPadula access control policy, as discussed by [6]. By recoverable we mean that write-ups accepted by the service are completed in the event of a system failure. By reliable we mean that, if a write-up is requested, the requestor can know if the write succeeded or failed, that is, acknowledgments are given to the writer.

We conclude this section with some definitions. In the following sections we review the Pump mechanism, define the basic write-up service, discuss necessary buffer size, present some usability enhancements, and discuss our conclusions.

In our discussion, we assume that all processes use *stable storage* in a recoverable way. Stable storage [2] is storage that is not affected by a system failure, e.g. disk storage. *Volatile storage* is storage that is affected by system failure; system failures cause the loss of possibly all of the contents of volatile storage. Stable and volatile are relative terms; we could consider off-line tape storage as stable and disk storage as volatile because disk hardware failures do not affect the off-line tapes. A more precise definition would distract us from our point. When we say that processes use stable storage in a recoverable way, we mean that they keep their data on stable storage, and follow the usual approaches to logging and caching in volatile storage [2] to ensure that their data can be recovered after a crash.

The Pump

The Pump provides communication from a low source process to a high destination process. It is a trusted mechanism with three components: trusted low buffer *TLB*, trusted high buffer *THB*, and communication buffer *CB*. A low source process sends

1. Yes, there is a problem with multilevel transactions that will be discussed in the conclusion.
2. Readers and writers being database system server/data manager instances.

a message to a high destination process by first passing it to the trusted low buffer *TLB*, which then gives the message to the communication buffer *CB*. When messages are in the *CB*, the trusted high buffer *THB* signals the high destination process and passes the message to it. Acknowledgments (*ACK*) and negative acknowledgments (*NAK*), and time-outs are used between *THB* and the destination and between *TLB* and the source. These acknowledgments are necessary for reliability and recoverability. They can be exploited as a covert channel because the destination process can modulate its *ACK* and *NAK* messages (or time-outs) to leak sensitive information to low. The Pump itself is trusted and cannot be exploited in this way. Figure 1 shows the Pump.

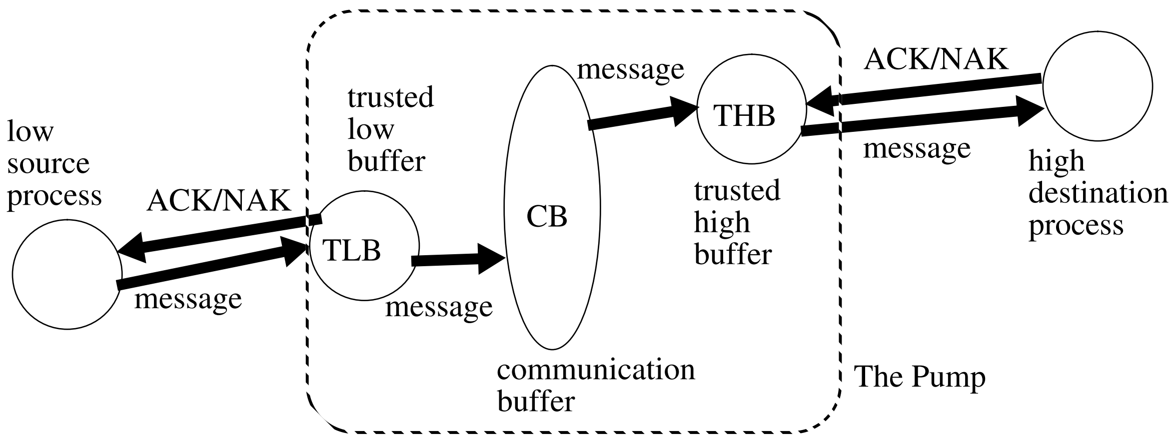

Figure 1. Message Passing From Low to High Using the Pump.

Kang and Moskowitz throttle the covert channel by delaying the acknowledgments from the high destination process in a way that gives approximately the same expected (mean) response time but significantly reduces the influence that the high destination process has on individual response times. The delay is added via a random variable with a modified exponential distribution. By computing a moving average they control the capacity of the channel and further complicate matters for Trojan horses.

The Write-Up Service

Now we look at a write-up service that does not incorporate a covert channel in its mechanism but nevertheless also provides effective reliability and recoverability. Like the pump, our write-up service also depends upon trusted software. The key point of the trust is that trusted software will only send legitimate control messages (i.e. *NAK* is only sent when a write-up fails). Protocol events caused by the write-up service are not due to a Trojan horse. We prevent modulation of the write-up service itself by disconnecting the flow of acknowledgments from high to low, and compensating for this by providing a probabilistic form of guaranteed delivery.

The service provides a set of write-up ports to the low process, that is, the writer. It provides a different set of receive ports to the high process that acts as the destination. The service maintains, in stable storage, a buffer to store the messages. The service follows a fairly conventional protocol, except there is no acknowledgment from the destination process to the source process:

The buffer slots can be either full or free and a message in the buffer can be removed from a receive-down port or overwritten by the write-up service. The low source process is allowed to query the write-up service regarding the status of a buffer slot, but not the status of a message in the buffer slot. The high destination process can query the status of buffer slots and messages in the buffer slots. The buffer starts with all slots free and no messages in the slots.

1. Low connects to a write-up port.
2. High connects to a receive-down port by specifying the kinds of messages it wants to receive.
3. Low sends a message. If the message is received by the trusted write-up service then an *ACK* is sent to low, the message is placed in a free buffer slot, the slot is marked full, and low may discard its copy of the message. If the message is not received by the write-up service then the trusted write-up service will either send NAK or low will time-out. In either failure case low retries the write-up. If the buffer is full, that is no free buffer slots are available, the write-up service will tell low to wait.
4. The write-up service signals or interrupts the high process to notify it that a message has arrived from low. After either a fixed or random time interval, the message's buffer slot is marked free. Freeing a buffer slot does not remove a message via a receive-down port.
5. High removes the message from its receive-down port. The write-up service does not tell low that the message has been removed from the port. Removing a message does not free its corresponding buffer slot.

To summarize, if we define a message in a free slot as discarded, denoting this condition as *dis*, removed from a receive-down port as *rem*, and overwritten as *over*, we have six possible message conditions:

$$dis \ \textbf{and not} \ rem \ \textbf{and} \ over \tag{1}$$

$$dis \ \textbf{and not} \ rem \ \textbf{and not} \ over \tag{2}$$

$$dis \ \textbf{and} \ rem \ \textbf{and} \ over \tag{3}$$

$$dis \ \textbf{and} \ rem \ \textbf{and not} \ over \tag{4}$$

$$\textbf{not} \ dis \ \textbf{and not} \ rem \tag{5}$$

$$\textbf{not} \ dis \ \textbf{and} \ rem \tag{6}$$

Steps three, four, and five can be repeated until either high or low decides to end the write-up session and disconnects. Flow control can be improved by overlapping several acknowledgments with a sliding window protocol. The low source processes are allowed to know how large the buffer is and when it is full, that is, they can be legitimately blocked when the buffer is full because the state of the buffer does not depend on the destination process. Figure 2 shows the components of the write-up service.

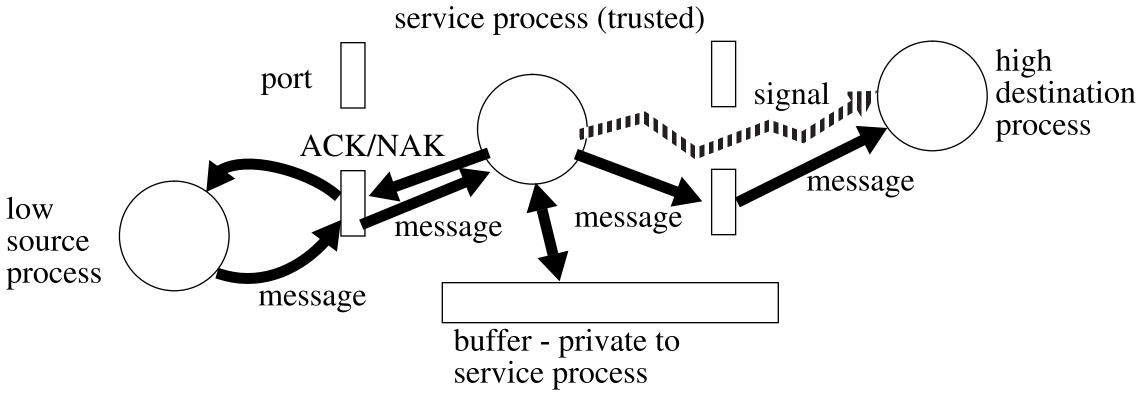

Figure 2. Basic write-up service

This protocol provides communication with conventional flow control between the source and the service process. We could even have an incremental improvement in the overall performance and reliability by having the service process send the messages (instead of a signal) and conduct a separate flow control protocol with the destination, as long as this protocol did not change the rate at which buffer slots were freed by the service process. The flow control is not modified by random extensions of the delay associated with sending a message. There is no covert channel due to acknowledgments sent from the high destination process to the low source process because there are none. If a malicious destination process refuses to receive messages, then the messages are overwritten[1]. Thus performance and security exceed that of the more general Pump mechanism. As we shall show next, the reliability and recoverability can be made arbitrarily good.

Big Buffers

Our write-up service depends on careful buffer management to avoid overwriting a message, condition (1) above. In normal operation and during short term failure, the success of our approach depends on being able to establish a *big* (enough) *buffer*. As we will show, it is possible to determine the probability of overwriting an update projection, as a function of the buffer size and system load. Because of this we can chose a buffer size that makes the buffer practically infinite.

Let us define a catastrophic failure κ as a failure of a database system that causes parts of some transactions to be lost and the database system to produce an incorrect history. This can happen even when correct transaction processing mechanisms are used because the failure (most likely a combination of failures) causes one of the underlying assumptions to be untrue (e.g. hardware failure or single-event upset in the running software). Because of the transaction processing mechanisms and the care taken in designing and implementing the system we expect the probability p_κ of catastrophic failure κ to be relatively small. Now define p_ω as the probability that an update projection will be overwritten. If the size L of the buffer is sufficiently large so

1. We make no claim to protect against denial of service, but such behavior would be detected quickly and the offending software removed.

that $p_\omega < p_\kappa$, then we say the buffer is a big buffer.

How big does a buffer need to be to be a big buffer? To answer this we model the destination process as a server in an M/M/1 queuing model [1], where the queue is finite. Recall that M/M/1 queueing models have exponentially distributed arrival and service rates, a single server, and are used to find steady-state values. The mean arrival rate of the service requests (write-ups) is denoted λ and the mean service rate (removal of messages by the destination process) is denoted μ. We call the ratio λ/μ the *offered load* (imposed on the system) and denote it by a. Offered load a represents the relative load on the system and is measured in units called erlangs. As a concrete example of how offered load a relates to performance we can find the delay for a particular offered load on our write-up system, using Little's Law [9]. Let L be the mean number of requests in a queue or in the server and W the mean length of time it takes request to pass through the system (i.e. sojourn time); then

$$L = \lambda W \tag{7}$$

for a wide range of queueing models, including the M/M/1 model with finite queue size.

For finite queues, the easiest way to apply Little's Law is to calculate the mean number of requests in the queue directly. Queueing theory [3] gives us the probability p_n of n update projections being present in the finite queue as

$$p_n = (1-a)a^n/(1-a^{max+1}) \qquad \text{for } 0 \le n \le max$$
$$p_n = 0 \qquad \text{for } n > max \tag{8}$$

where max is the size of the buffer. We then compute the mean number of requests in the queue as $L = \sum_{0 \le n \le max} n \cdot p_n$.

So, if our write-up system was receiving one update projection per second on the average (i.e. λ=1.00), the buffer size was 600 update projections, and the offered load was a=0.99 erl, then, by Little's Law, a write-up would take roughly two and a half minutes to propagate, on the average. Practical systems operate with much smaller offered loads; for example if we take a=0.5 erl, the update projection propagates in about one second. These values would hold even in an untrusted system that could use conventional flow control protocols.

If we set $n = max$ in equation (8), we get p_{max} the probability of a full buffer. Since a full buffer causes an overwrite, we can treat p_{max} as p_ω probability of overwriting an update projection. Figure 1 shows a plot of buffer size as a function of offered load,

1. Besides being tractable, this model is appropriate because the source and destination processes in a replicated architecture are essentially the same, though possibly loaded differently. With respect to tractability, our current model of a finite M/G/1 queue must be run overnight to compute a single data point. Its results tend to agree with the more tractable M/M/1 model.

for a range of overwrite probabilities.

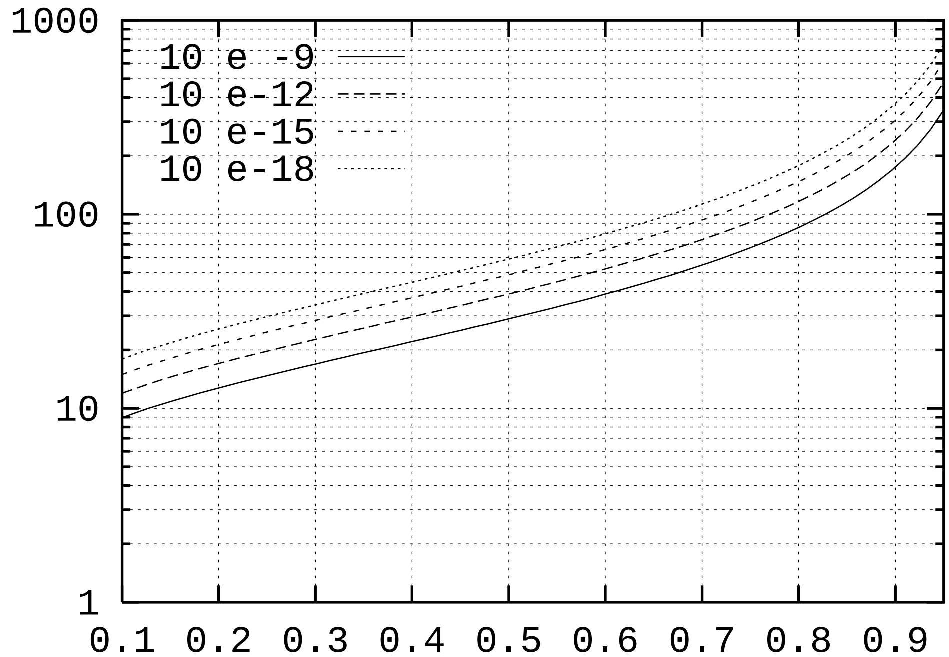

Figure 3. Buffer size as a function of offered load for overwrite probabilities of 10^{-9}, 10^{-12}, 10^{-15}, and 10^{-18}.

As we see from Figure 3, a buffer size between 100 and 500 is sufficient for offered loads as high as a=0.95 erl with a overwrite probability of 10^{-12}, a condition where the average delay for our previous example is about 28 seconds. Since there are 3.1536×10^7 seconds in a year, it is unlikely that our write-up system will have an overwrite during its useful lifetime. Significant increases in reliability can be obtained for relatively small increases in buffer size. If we reduce our overwrite probability to 10^{-15}, we only need a buffer size of about 600 at a offered load of a=0.95 erl.

Since update projections are relatively small objects (an average size of 1K bytes is quite generous for a logical update projection[1]) provision of big buffers is practical. In practice the average size of an update projection is likely to be an order of magnitude smaller. Even if our update projections were 1K bytes, we would only need about 600K bytes of buffer storage for a write-up service.

Because buffer exhaustion cannot be used to communicate, we assume no attempt by the untrusted sender or receiver to fill up the buffer in order to cause an unauthorized information flow. This justifies our use of conventional models based on independent arrival and service times, and conventional steady state values.

Recoverability

From the perspective of the source processes and the write-up service proper, the

1. A logical update projection is implemented by sending the text of an update transaction rather than the physical writes it generates.

write-up service appears to handle system failures just as a conventional system would. Messages in the buffer are in stable storage; transactional logging procedures can be used to restore the buffer in the event of a system failure. If source processes use similar techniques, they can retransmit messages that were not acknowledged by the write-up service. For this reason, the write-up service provides recoverability for failures of the source processes and of the write-up service itself; we will not discuss it further .

The question of recoverability with respect to failure of the destination process is more interesting. Our basic approach is to make the write-up service buffer large enough to hold all the messages that may be sent before a destination process can recover . Here we are only able to succeed because we restrict the problem to replicated-architecture database systems. Because we are using source processes with finite memory we know we will have to chose to discard some update projections (write-up messages) in the event of a long-term destination process failure. This choice has nothing to do with write-up strategies but rather with the finite capacity of the source process. The source must continue to process new updates at its own security class and it must eventually run out of space to store the new update projections it wishes to propagate and so must discard some of them. This is not a problem; the same choice is made for conventional distributed database systems [2, § 8.5]. For this reason, if we restricted ourselves to use of the Pump, we would still have to discard some write-up projections in the event of long-term failure [1].

Since we know some update projections will have to be discarded in some cases, we can chose to define a short-term failure to be one that fits our desired range of offered loads. That is, if we expect offered load a to be small and failures to be infrequent, we can define "short" as a longer period of time than if we expect frequent failures of the destination process or if we expect offered load a to be relatively large. In any case, in determining the required buffer size, we simply treat short-term failures as additional write-up requests that tie up the system for some period of time equal to the time needed to detect and recover from the failure.

Usability Enhancements To The Basic Service

There are some non-critical enhancements we can make to the basic service to improve its usability in practical systems: message time-outs, overwrite priorities, and variable buffer sizes.

First, we can make it easier for the destination process to manage its rate of message receipt and the write-up service to adjust its buffer size. To do this, we set a timer for each message when it is accepted by write-up service. Messages received for write-up are stored until they either expire or they are received by a high destination process. If a message is received it is marked as such but is not removed from the buffer . Only high destination processes can tell if a message has been received. The time-out period for messages is fixed at system generation time. Upon time-out the message expires, is marked as such, and it *may* be overwritten or discarded by the write-up service. An expired message may be received and a received message will expire.

1. Recall that one-copy serializability does not require all writes to update all copies.

Only expired messages are overwritten or discarded. A human user (database administrator or system security officer) can monitor the performance of the destination process with respect to the time-outs and adjust the buffer size of the write-up service if necessary . If we use this option, we want to hide the buffer size from the source process.

A second enhancement we can make will improve the ability of the write-up service to ensure that critical messages are more likely to be received. The write-up service can provide a priority parameter that indicates the criticality of the message. If an overwrite is necessary , the lower priority messages will be overwritten first.

A third enhancement we can make is to provide variable buffer size. The write-up service does not need to maintain a fixed buffer size, if the buffer size is not visible to the sending processes. W ith this approach, the write-up service would maintain an estimate of offered load a and adjust the buffer size as needed. In the case of a full buffer , the write-up service would first try to expand the buffer and then overwrite an earlier message if no more free space were available. Our model shows that this is possible, since big buffers will fit easily into the stable storage space available on a dedicated frontend or replica controller .

It is also possible to let the source process know the size of the buffer used by the write-up service but still vary the effective buffer size. If the destination process is designed as an interrupt handler (i.e. a small program that quickly removes data from a port and then schedules work for a larger process that uses the data) it can have a variable size buffer . This second buffer can be implemented at the destination security class and thus will be invisible to the source process. The destination process buffer can vary according to a, and the destination process will only be responsible for receiving write-ups from the service. The replicated architecture database system can then accept update projections from the destination process.

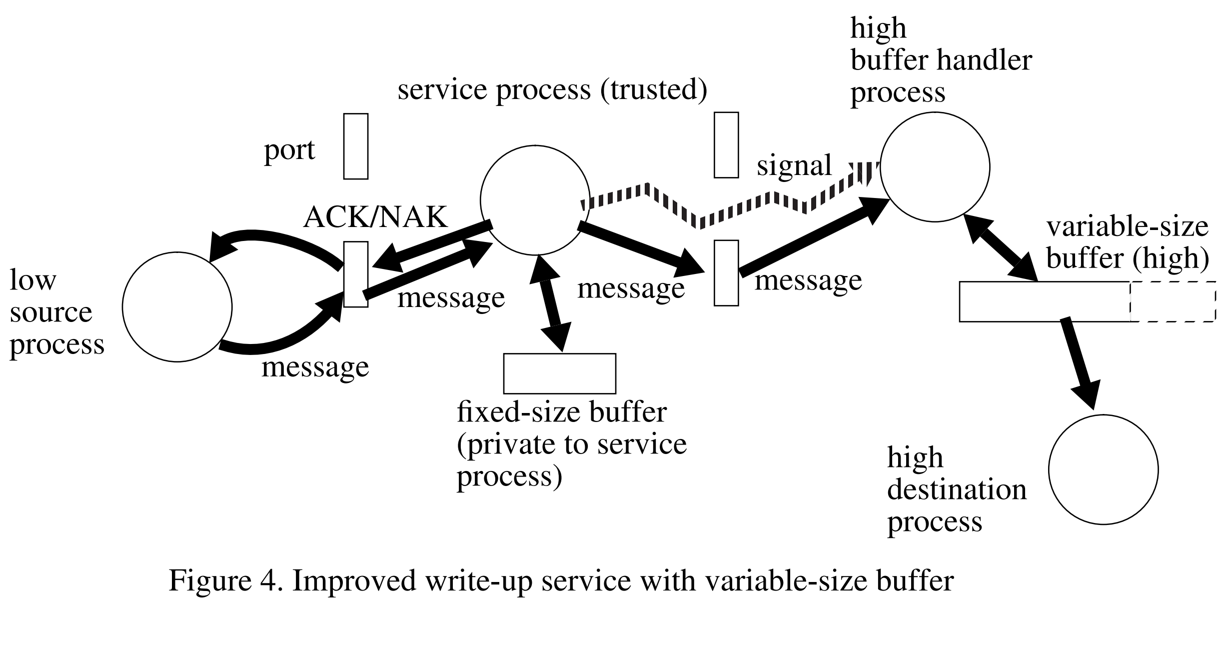

Figure 4. Improved write-up service with variable-size buffer

Conclusions

The Pump represents a good general mechanism for writing up. However, we believe that practical covert-channel-free alternatives exist for the special case of the replicated architecture, with comparable or better time performance without a meaningful sacrifice of reliability and recoverability. The write-up service we have described here is one alternative. Our alternative mechanism has the same performance as an untrusted communication mechanism, that is, no delays are introduced. It has no covert channel due to acknowledgments. There is a nonzero probability of overwriting a message, but the reliability and recoverability can be made arbitrarily good by appropriate choice of buffer size. Acknowledgements and flow control can be extended to cover, separately, both source-to-service and service-to-destination communications. We can easily make the write-up mechanism more reliable than the system it supports. It is not clear that an adjustable covert channel can be set to be smaller than the smallest covert channel that might be exercised, particularly in light of the small message criterion of Moskowitz and Kang. On the other hand, we must admit that our write-up service is not general, but limited to an important special case and may not apply to other special cases.

Stable storage is inexpensive compared to the cost of developing new applications on high-assurance trusted systems. This is the same justification for the replicated-architecture approach, which is the place we expect this service to be used. Where the offered load a is less than 1.1 erl, we can provide the desired reliability within the bounds of conventional disk systems. In the more likely case, where the offered load a is less than 0.95 erl, we can succeed with buffers whose size is between 10^2 and 10^3 update projections.

Our write-up service has not been specified so that it can deal with creation of new data items at higher security classes. This kind of operation is not available in the current SINTRA prototype [7], but is necessary for fully general multilevel transactions [4]. The problem of resource exhaustion by unbounded writing into finite high storage is difficult. We can provide for this service by setting quotas on the creation of new data items via blind write-up, but this seems less than satisfactory. Future work should investigate models and mechanisms for extending big buffer write-up to handle creation of new data items in a more elegant fashion. We also plan to look at more advanced models of buffer size, such as M/G/1 queues with finite buffers.

Acknowledgments

Carl Landwehr suggested several improvements to the paper. Our queueing theoretic model (both on paper and in Mathematica) has benefitted from several discussions we had with Ira Moskowitz. Insightful comments by anonymous referees also improved this paper.

References

1. BELL, D. and LAPADULA, L. Secure computer systems: unified exposition and Multics interpretation. MTR-2997, MITRE Corp., Bedford, MA, 1975.

2. BERSTEIN, P., HADZILACOS, V., and GOODMAN, N. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

3. COOPER, R. *Introduction to Queueing Theory*, North-Holland, 1981.

4. COSTICH, O. and JAJODIA, S. Maintaining multilevel transaction atomicity in MLS database systems with kernelized architecture. in *Database Security VI: Status and Prospects*, B. Thuraisingham and C. Landwehr, ed. North-Holland, 1993.

5. COSTICH, O., McLEAN, J., and McDERMOTT, J. Confidentiality in a replicated architecture trusted database system: a formal model. in *Proceedings of the 1994 Workshop on Computer Security Foundations*, Franconia, NH, June 1994.

6. FROSCHER, J. and MEADOWS, K. Achieving a trusted database management system through parallelism. in *Database Security II: Status and Prospects*, C. Landwehr, ed. pp. 151-160, North-Holland, 1989.

7. KANG, M. FROSCHER, McDERMOTT, J., COSTICH, O., and PEYTON, R. Achieving database security through data replication: the SINTRA prototype. NRL TM 5400-041A. February 1994.

8. KANG, M. and MOSKOWITZ, I. A pump for rapid, reliable, secure communication. *1st ACM Conference on Computer and Communications Security*, Fairfax, Virginia, November 1993, pp. 119-29.

9. LITTLE, J. A proof of the queueing formula $L=\lambda W$. *Operations Res.*, 9, 3, 1961, pp. 383-387.

10. MATHUR, A. and KEEFE, T. The concurrency control and recovery problem for multilevel update transactions in MLS systems. in *Proceedings of the 1993 Workshop on Computer Security Foundations*, pp. 10-23, Franconia, NH, June 1993.

11. FROSCHER, J., KANG, M., MCDERMOTT, J., COSTICH, O., and LANDWEHR, C. A practical approach to high assurance multilevel secure computing service. submitted for publication.

12. MOSKOWITZ, I. and KANG, M. Covert channels - here to stay?, to appear in proceedings of COMPASS '94.

13. R. SANDHU, R. THOMAS and S. JAJODIA. Supporting timing-channel free computations in multilevel secure object-oriented databases. in *Database Security V : Status and Prospects*, C. Landwehr and S. Jajodia, eds., pp. 297-314, North-Holland, 1992.

14. R. THOMAS and R. SANDHU. Implementing the message filter object-oriented security model without trusted subjects. in *Database Security VI: Status and Prospects*, B. Thuraisingham and C. Landwehr, eds., pp. 15-34, North-Holland, 1991.