This paper is copyrighted by The Institute of Electrical and Electronics Engineers, Inc. (the IEEE) and it is published in the Proceedings of the 29th Fault-Tolerant Computing Symposium (FTCS-29), IEEE Computer Society Press, pages 56-63, Madison, Wisconsin, USA, June 15-18, 1999.

Incremental Messages: Micro-Kernel Services for Flexible and Efficient Management of Replicated Data

Carlos Pérez
Dpt. de Informática y Electrónica
Universitat de València
C/ Doctor Moliner, 50
E-46100 Burjassot SPAIN
carlos.perez@uv.es

Germán Fabregat
Dpt. de Informática
Universitat Jaume I
Campus del Riu Sec
E-12071 Castellón SPAIN
fabregat@inf.uji.es

R.J. Martínez & G. Martín
Institut de Robótica
Universitat de València
Polígono de la Coma S/N
E-46980 Paterna SPAIN
rafael.martinez@uv.es
gregorio.martin@uv.es

Abstract

Incremental messages have been designed to efficiently and flexibly manage replicated copies of critical data. We describe this new type of messages and show how applications benefit from an extended message passing interface which provides support for creating, updating and recovering data copies, combining the advantages of both kernel and user-level approaches. The paper describes the interface offered to applications, its functionality, and how it can be used to implement traditional fault-tolerance mechanisms such as passive replicas, checkpointing and recovery. A fault-tolerant transactional service that exploits the advantages of incremental messages is also presented. Furthermore, the paper presents details about the implementation of incremental messages in VSTa, a micro-kernel based operating system, and the first experimental results obtained.

1. Introduction

Checkpointing and rollback recovery have been traditionally used to tolerate errors, taking advantage of the natural redundancy that distributed systems have. Considerable research efforts have been addressed to reduce as much as possible the inevitable overhead that this technique introduces. Two complementary ways of reducing checkpointing overhead are to optimize checkpointing code, and to reduce the amount of data that has to be included in every checkpoint. Researchers have applied both of them following two different approaches. The first one consists in including checkpointing code in the operating system kernel and employing memory mapping techniques (such as copy-on-write or incremental checkpointing) to make it efficient [1, 2]. The second approach is to implement checkpointing services as user-level routines, which can then be tailored to copy only the critical data of the application [3, 4, 5]. Although both approaches are complementary, there is no implementation that combines their advantages, since current operating system kernels do not provide suitable support for implementing checkpointing and rollback recovery at the application layer. As it is suggested in [6], incremental and concurrent checkpointing, which are effective methods of reducing the overhead of checkpointing, should be provided by the operating system kernel.

In this scenario, our work has focused on the design of a reduced set of services that can be incorporated to operating system kernels to provide satisfactory support for user-level checkpointing and recovery [7]. These services have been designed specifically for micro-kernels, because they (micro-kernels) enforce modular design, appropriated for highly dependable systems [8, 9], and because most of current distributed operating systems are based on the micro-kernel concept, with classical operating system services provided by user-level processes. (Chorus [10], Amoeba [11] and Mach [12] are good examples of it.) Even when traditional kernels are used, numerous distributed services are nevertheless implemented by user-level code.

In this paper, this new set of kernel services —which we have called "Incremental Operations" after incremental checkpointing— are introduced. All of them are based on the same basic mechanism and therefore the increment to the kernel size and complexity is small. At the same time, these services are versatile enough to support the implementation of different user-directed checkpointing and recovery algorithms, as well as other fault-tolerant mechanisms as passive replicas or transactional services. It is worth noting that incremental operations are designed for versatility, and not for transparency, since traditional kernel checkpointing services are appropriate for this purpose.

The outline of the paper is as follows. Sections 2 and 3 describe Incremental Operations, focusing on their functionality and their implementation, respectively. Then, section 4 shows several examples of how they can be used to

develop fault-tolerant applications. Section 5 presents the first experimental results obtained after the implementation of these services in VSTa [13], a micro-kernel based operating system. Finally, conclusions of the work and future research directions are presented in Section 6.

2. Incremental Operations

Incremental Operations offer to upper layers' processes tighter control over both their own address space and the address space of other processes. This is achieved using the support of the memory management unit (MMU); facilitating efficient management of replicated information. In addition, they follow the inter-processes message-passing model, thus easing the implementation of fault-tolerance techniques based on that model.

2.1. Incremental Messages

Incremental Messages (IMs) share with traditional messages the function of transferring data from the sender's to the receiver's address space. However, they have been designed to update data copies maintained by different processes. For this reason, they present two unique features: (1) only those pages from the virtual memory region specified by the sender that have been modified since the last IM are transferred to the receiver, and (2) the destination of the message is an existing virtual memory region of the receiver's address space, which contains a copy of the sender's one. Both sender (P1 in Figure 1) and receiver (P2) must reside in the same node, and the effect of sending an IM on their address spaces is depicted in Figure 1. Initially, P2 holds a back-up of P1's critical data that is updated with the modified pages from P1.

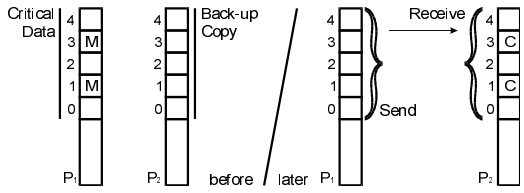

Figure 1: Incremental messages.

IMs are a simple and efficient mean to update data copies maintained in the same node that the critical data they back-up. They are easy to use because to update a back-up copy (or copies), the process holding the primary copy of critical data just has to send an IM to the process (or processes) holding the back-up copy (or copies). They are efficient because only critical data that have been modified have to be transmitted. This efficiency can yet be increased if COW is used.

2.2. Other Incremental Operations

The same micro-kernel mechanism that is needed to implement IMs can be used to provide support for other fault-tolerance mechanisms. The following subsections describe these derived services.

2.2.1. Negative Incremental Messages (NIMs). NIMs have been designed to allow a process to simply and efficiently discard the changes made in the critical data since the previous IM. Their use, clearly oriented to perform efficient rollbacks, is described in the next paragraph.

As in the example of Figure 1, let us assume that P1 has a working copy of the critical data, while P2 holds its back-up. If P1 decides to discard the changes made on its data, comprising only pages 1 and 3 in the figure, it sends an IM with the *negative* option to P2. The micro-kernel does not include the modified pages in the message, but an indication of these pages. P2 then receives the message and identifies it as a NIM, using its contents as a map to select the pages to be returned to P1. A second message is sent back from P2 to P1 using the *map* option (with send) to indicate to the micro-kernel that the pages specified by the map must be included in the IM (instead of those modified by P2, as it would happen with a normal IM). When P1 receives this last message, the micro-kernel replaces the modified pages 1 and 3 with those sent by P2, thus effectively discarding the changes P1 had made.

2.2.2. Packed Incremental Message (PIM). A PIM is a compact representation of the information present in an IM. In addition to the actual pages to be transferred, a PIM contains a special header called 'map', which specifies the contents of the original IM. The map contains the starting address and length of the source region, as well as a list of the pages included in the message.

The main objective of PIMs is to improve the management of IMs by processes that do not hold a copy of the sender's critical data, such as intermediate processes between sender and receiver, or processes handling secondary storage. As an example of how PIMs ease efficient handling of IMs by intermediate processes, Figure 2 shows the case of inter-node communication.

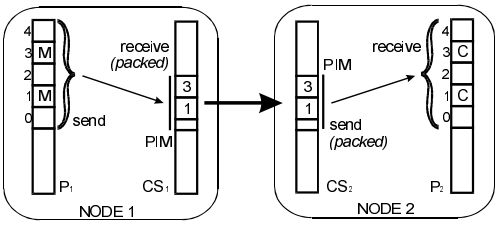

Figure 2: Transferring IMs between nodes using PIMs.

As Figure 2 illustrates, IM packing and unpacking is carried out when the *packed* option is used with send and/or receive. If this option is used when receiving an IM, the micro-kernel adds a new memory region to the virtual address space of the receiver, mapping the whole PIM (both the map and the pages it contains) as a consecutive range of logical addresses. The effect of using the *packed* option when sending an IM is that the micro-kernel interprets the virtual memory region as a PIM (instead of building the IM with the modified pages of the region). If

the message is not received with this option, then the pages included in the IM substitute their counterparts in the receiver's address space, just as it happened with IMs. If the *packed* option is used, then the IM is received as a PIM, as it has just been described before. Finally, when the *packed* option is used when both sending and receiving the IM, the data is copied from the sender's to the receiver's address space as a conventional message.

2.2.3. Incremental Copy (IC). This service is designed to manage replicated data when both original and copy reside on the same address space. Its basic operation is the same as that of IM, except that it transfers data between two memory regions of the same process. As with IMs, a Negative Incremental Copy (NIC) has also been defined to allow simple and efficient rollback.

2.3. Header Management

Although IMs have been designed to support the management of replicated data, other requirements of fault-tolerant applications should also be considered. In particular, when using modular redundancy, tasks such as voting, selection or reordering of messages are performed either by library code at senders or receivers, or by intermediate processes [14, 5, 15, 2, 16]. Hence, it is important that IMs can be handled efficiently by these intermediate agents. Furthermore, reordering of messages by library code is not possible with IMs as they have been described so far, since the data copy is updated when the message is received.

To cope with these requirements, the *receive* primitive includes the possibility of receiving only message headers while out-of-line data [18] are placed in an auxiliary queue managed by the micro-kernel (which offers appropriate services to let processes access the contents of messages in the queue: *map*, *forward* and *delete*). The information present in this header can vary from one implementation to another, but should include the in-line data of the message and the list of pages included in it.

This feature, in addition to support the management of intermediate messages, is a mean for error detection and confinement (see subsection 4.1.) and, in the case of using intermediate agents to handle messages, it avoids mapping out-of-line data, thus gaining in performance and protecting them against errors in intermediate agents.

2.4. Management of Multiple Destinations

When several copies of critical data are to be hold, (because more than one simultaneous fault must be tolerated, for example) not all of the copies are to be updated simultaneously. (See subsection 4.2.) So that IMs can be used in these cases, the micro-kernel should keep track of modified pages with regard to every possible destination of IMs (where each copy is maintained). This requires that the concept of "modified bit" (as it is provided by MMUs) be extended into a bitmap managed by the micro-kernel. From this point on, we would refer to this bitmap as "the modified bitmap", after "modified bit". Each element of this bitmap represents the state of the page (modified or not) regarding one possible destination of IMs.

Since user-level code must be able to establish when each copy must be updated, Incremental Operations include a parameter, called *incremental destination*. The micro-kernel uses this parameter to find out which element of the modified bitmap should be considered to establish the set of modified pages.

3. Implementation Issues

Despite the versatility of IMs, the interface that needs to be provided to applications is very compact because IMs are just an extension to the standard message-passing interface. Apart from header management and incremental copy, the whole set of services described in section 2 has been implemented in the micro-kernel based VSTa operating system [13], integrating IMs into its original message-passing interface.

3.1. Basic Interface

The interface can be better described in terms of a basic message passing interface, comprising only the services *receive(src, msg)* and *send(dest, msg)*. Here, *src* and *dst* are any valid source and destination in the existing OS, and *msg*, for the purpose of exchanging IMs, must include the following fields: *flags*, *data*, *map* and *msgid*. The meaning of this fields depends on the operation (*send* or *receive*) and the type of IM being exchanged. Tables 1 and 2 describe these fields for both operations.

Table 1: Message fields when sending an IM.

Field	Meaning
flags	Establishes sending mode. Valid flags are: **MAP**: sender includes the list of pages, which should be included in the message. **NEG**: only the map must be included in the message, no pages must be actually transferred. **PACKED**: *data* is a PIM
idst	Incremental destination (see subsection 2.4).
data: va, len	Virtual address, *va*, and length, *len*, of source memory area. It must point at the critical data area if PACKED is not set, and to a PIM otherwise.
map: va, len	It must point at a map of pages (usually obtained from a previous NIM).

Table 2: Message fields when receiving an IM.

Field	Meaning
flags	Establishes receiving mode. Valid flags are: **PACKED**: IM must be packed at reception. **HEADER**: Only IM's map must be mapped onto the receiver's address space. **SRCVA**: Use virtual address from source. The kernel updates this field to signal the receiver the type of IM received, setting NEG when appropriate.
data: va, len	If no flag is set, it must point to the memory area where updates must be applied. When SRCVA is set, the kernel returns the *va* and *len* included in the IM, pointing at the memory area where updates have been applied. When PACKED (or HEADER) is set, the kernel returns the initial address that the receiver can use to access the PIM (or IM's map), and its length. If the IM is a NIM (flag NEG set by the sender), then the kernel returns the address of the IM's map as above.
msgid	When HEADER is set, the kernel returns an identifier that lets the receiver invoke operations on the message while it is in the auxiliary queue. (See subsection 2.2.3.)

3.2. Integration into VSTa's interface

VSTa's message passing interface is client-server oriented, providing three basic services: *msg_send*, *msg_receive* and *msg_reply*.[1] Clients can control data flow by means of the M_READ flag. When set, the client will receive data provided by the server in its call to *msg_reply*; otherwise, data sent by the client will be transferred to the server at *msg_receive*. (Since VSTa's interface is connection oriented, it provides appropriate additional services for handling connections, which need not be modified for supporting IMs.)

For IM exchange, the arguments that clients and servers must provide when invoking *msg_send*, *msg_receive* and *msg_reply* depend on the information flow. If a client does not set M_READ when invoking *msg_send*, then it is acting as the sender and the meaning the fields of the message are those of Table 1. However, if it sets M_READ, then those of Table 2 apply. In any case, both client and servers must set M_INCR to flag the kernel that they are sending or waiting for an IM.

In subsection 2.2, where NIMs are introduced, it is described how to exchange NIMs using two messages: the first one to request the modified pages and the second, the reply, with the requested pages. With our implementation of IMs on VSTa, only one client-server interaction is needed to exchange a NIM. The client sets NEG and M_READ in *msg_send*. Then the server receives, by means of *msg_receive*, the map of modified pages (and an indication that the IM is negative, as it is described in Table 2) and sends the requested pages in the reply, setting MAP in the call to *msg_reply*.

One of our aims when integrating IMs into VSTa was to permit that any server could receive both normal and incremental messages from the same port. We have achieved this objective providing the necessary automatic conversions when client and server do not agree on the type of message being exchanged. The R process, which is described in subsection 4.4, exemplifies this point.

3.3. Dirty Bit Management

When extending VSTa's message passing interface to support IMs, one important problem that we have addressed has been how to share the MMU's dirty bit with existing kernel modules. The solution should let the kernel maintain several independent copies of this bit per page, so that each page could be sent to several different destinations at different rates (see the discussion about management of multiple destinations in subsection 2.4).

To cope with these requirements, for each page the kernel maintains V, a bitmap of length N with each bit representing one potential destination for IMs (incremental destination). Besides, one more bit is kept for each page, which we will call D', which is the traditional dirty bit maintained to flag if the page needs to be written to disk in case of being stolen. Finally, let us call D the MMU's dirty bit. The algorithm used is as follows:

When a page is the source of an incremental operation, the kernel:

1. If D is set, it sets both D' and every bit in V, and then clears D.
2. Checks V_i (being i the incremental destination for the operation) and, if it is set, includes the page in the operation. Then, it clears V_i.

When a page is the destination for an incremental operation, the kernel sets every bit in V (because it has changed and, therefore, should be included in the next incremental operation) and clears D. D' should be set only if it should be written to disk in case of being stolen.[2]

When the kernel steals a page:

1. Checks D. If it is set, then it sets both D' and every V bit. Then clears D.
2. Checks D'. If it is set, then writes the page to disk and clears D'.

4. Using Incremental Operations

Incremental Operations are a useful tool for the implementation of fault-tolerant applications. The following subsections show how they can be used in different situations, easing the design of several fault-tolerance mechanisms and improving their performance.

4.1. Error Detection and Confinement

It has been outlined how Incremental Operations offer features that can be utilized to increase the reliability of applications. These features can be used by user-mode layers when applications have special dependability requirements.

As it will be shown, the possibility of receiving the header of a message before its contents are mapped onto the receiver's address space is an efficient way to detect sender errors and confine their effects. If the receiver has information regarding the set of pages the sender has permission to modify, the information included in an IM header can then be used to detect errors that have caused the process to write on an unexpected page. Experimental evaluation of standard error detection and confinement mechanisms shows that the checks performed by the MMU when translating logical addresses are very effective to maintain the fail-silent behavior in computers without error masking [19].

[1] Actually, there is another service, *msg_error*, which can be used by server processes to flag error conditions to their clients and, therefore, it can be considered as an special case of *msg_reply* for our discussion.

[2] We have implemented IMs for zero-fill-on-demand virtual memory regions. Therefore, if the source page was not filled yet we just mark the destination as not valid, not swapped, not modified; so that it be filled (with zeroes) when it be referenced.

The example illustrated in Figure 3 shows an application consisting of two processes that concurrently access their private copy of the same data using a distributed mutual exclusion algorithm to avoid non-consistent accesses. Header management (see subsection 2.2.3) lets the receiver detect errors in the sender before they are propagated to its (the receiver's) address space.

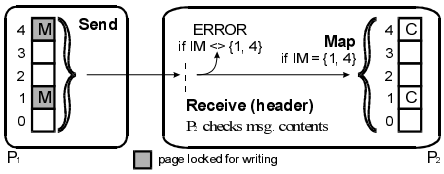

Figure 3: Using the possibility of obtaining only the header of an IM to improve error detection and confinement.

4.2. FT Services Through Server Groups

Another example that demonstrates the effectiveness of IMs to build fault-tolerant applications is the implementation of fault-tolerant services through loosely coupled server groups, following the description given in [3]. There, each group includes a primary server that interacts with clients and periodically updates the state of the remaining processes in the group.

IMs allow the simple and efficient implementation of that technique. To update the state of several replicas the primary server has only to send an IM to each of them, relying on the management of multiple destinations by the micro-kernel. Replicas should only use the receive service to keep their copies up to date. The use of IM guarantees that only modified pages are sent in each message, thus reducing the amount of actual data transferred as the frequency of the updates increases. Therefore, the use of IMs allows keeping the replicas closer to the service state, reducing the time required for recovering after failure of the primary server.

The primary server may update each replica with different periodicity. To accomplish this, it just have to use one different *incremental destination* (see subsection 2.4) for each replica. Figure 4 shows an example with a primary server and two replicas (0 and 1) associated (by the primary server) to incremental destinations 0 and 1. Each page of the server is depicted with its state ('M' meaning modified and '-' not modified) regarding both incremental destinations (to simplify the figure, only these two elements of the "modified vector" are shown). The micro-kernel keeps this information updated transparently to higher system layers. The primary server has just sent an IM to replica 0 and so no page is modified regarding this destination. At the same time, the log kept by replica 0 is empty. On the other hand, replica 1 has several entries in the log, and thus several pages of the primary server are modified regarding destination 1.

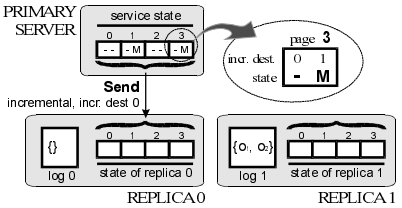

Figure 4: FT through loosely coupled server groups.

4.3. Transactional Services

The versatility and efficiency of Incremental Operations have been also demonstrated by means of the implementation of a transactional file server, that allowed us to obtain some of the first experimental results, which are shown in the following section. This subsection describes the implementation of transactional services [17] based on incremental operations.

The structure of the fault-tolerant transactional service is depicted in Figure 5, for two ongoing transactions. Data regions holding critical data are shown in dark gray. Region D holds service state, D_R is a back-up copy of D, and C_W^i is the working copy of D for transaction i. The service is provided by several cooperating processes. Process TS keeps the original data copy (D) and coordinates the transactions; process R acts a back-up server to TS and a set of T_i processes (one per transaction) interact directly with the clients of the service. Each T_i process uses the services provided by TS to coordinate the operations of its transaction with those from other T_i processes.

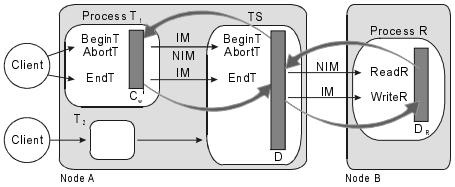

Figure 5: Structure of the service for two concurrent transactions.

When a client connected to a T_i process starts a transaction (service *BeginT* of T_i), C_W^i is updated from D through an IM from TS to T_i. During transaction i, all operations are performed (by T_i on client's behalf) on this working copy (C_W^i). If the transaction is to be committed (service *EndT*), D is updated through an IM from T_i to TS. However, if it has to be aborted, changes to C_W^i are discarded by means of a negative IM (NIM) from TS to T_i. To achieve fault-tolerance, TS keeps D_R updated using the services provided by the back-up server process (R). When the service is started, TS sends an initialization request to R, causing that D contents be copied into D_R. From then on, TS sends an IM to R (service *WriteR*) each time a T_i process invokes TS's service to commit the transaction. Only when D and D_R have been updated, R sends back an acknowledgment to T_i. If the transaction is to be aborted, no interaction between TS and R occurs, as D (and thus D_R) is not modified.

Although this service supports concurrency, we have deliberately omitted details about concurrency control algorithms. The reason for this is that the proposed structure is not dependent on the algorithm used, supporting both optimistic and pessimistic approaches.

4.4. Detailed IM Programming Example

In this subsection, we show in detail process R, which has just been described as part of the fault-tolerant transactional service. This process holds a back-up copy of critical data for one client process at a time. Figure 6 shows its main loop, where client messages are received and processed. In the picture, connection-oriented message handling has been omitted because it is not significant to illustrate IM programming. Error handling has also been omitted to simplify the example.

When the client starts, it should invoke FS_OPEN (sending a normal message with FS_OPEN in *msg.m_op*). As a result, process R prepares a new memory region (invoking *mmap*), which will then be used to keep the copy of client's critical data. Please note that the server invokes *msg_receive* with the M_INCR flag cleared until the back-up region is created. From this point on, the server sets M_INCR and specifies as destination for IMs the newly created memory region. The client can update the back-up copy sending an IM with FS_WRITE in *msg.m_op*. To satisfy this request, the server just has to reply to the client, because the back-up copy is updated by the kernel at message reception. Otherwise, the client can discard the changes made to the original data just sending a negative IM. Then, the server receives a new memory region whose address and length are contained in *msg.m_seg[SEG_PGS]* (the values provided by the server to the kernel when invoking *msg_receive* are simply discarded by the kernel, as a new region —with its own virtual address range— is begin provided to the server). To satisfy the request, the server just has to fill *msg.m_seg[SEG_MAP]* and *msg.m_seg[SEG_PGS]* with appropriate values (the map is the new region and the source data pages are those of the back-up region created at FS_OPEN) and then invoke *msg_reply* with flags M_INCR|M_MAP. After replying, the server discards the region holding the map, which is no longer needed.

```
port_t rootport;        /* Port for incoming msgs. */
void *datap = NULL;     /* Start addr. of back-up region */
uint datapgs = 0;       /* Region size (in pages) */

static void bkup_main()
{
      struct msg msg;
      void *map;

loop:
      if (IMs are welcome) {
            /* IM: set destination for FS_WRITE */
            msg.m_nseg = 1;
            msg.m_seg[SEG_PGS].s_buf = datap;
            msg.m_seg[SEG_PGS].s_buflen = ptob(datapgs);
      }
      imsg_receive(rootport, &msg,
                  (IMs are welcome)? M_INCR : 0);
      switch (msg.m_op & M_OPMSK) {
...
      case FS_OPEN: /* m_arg = sz. of crit. data */
            datapgs = msg.m_arg;
            datap = (void *)mmap(0, ptob(datapgs),
                        PROT_READ|PROT_WRITE,
                        MAP_ANON|MAP_PCOW, 0, 0);
            /* Empty answer to client */
            msg.m_nseg = msg.m_arg = msg.m_arg1 = 0;
            msg_reply(msg.m_sender, &msg);
            break;
      case FS_WRITE:          /* Incr. message */
            /* Empty message to client */
            msg.m_nseg = msg.m_arg = msg.m_arg1 = 0;
            msg_reply(msg.m_sender, &msg);
            break;
      case FS_READ:           /* NEG Incr. message */
            /* Client sends us MAP in PGS */
            msg.m_seg[SEG_MAP] = msg.m_seg[SEG_PGS];
            map = msg.m_seg[SEG_MAP].s_buf;
            /* Return to him what he just asked for */
            msg.m_seg[SEG_PGS].s_buf = datap;
            msg.m_seg[SEG_PGS].s_buflen = ptob(datapgs);
            msg.m_nseg = 2;
            imsg_reply(msg.m_sender, &msg, M_INCR|M_MAP);
            /* Free map region */
            munmap(map, 0);
            break;
...
      }
      goto loop;
}
```

Figure 6: Detailed specification of R process.

This example shows how easily IMs can be used to provide back-up and recovery of critical data at user-level. Although process R could be extended to support more than one client simultaneously, each one using several back-up regions, back-up and recovery is simpler to implement when critical data is kept in one memory region. The simplest approach is to modify existing dynamic memory management routines, so that the application can transparently use standard services, such as *malloc* or *free*. This is the approach we followed in the initial phases of implementation to build applications to test IM services.

5. Experimental Results

The first experimental results obtained after implementing IMs show that Incremental Operations can be used in real applications. However, we do not claim to have a particularly efficient implementation of IMs. Previous published results from other researchers [1, 6] prove that incremental and copy-on-write techniques significantly improve performance, and the strength of IMs is to offer these kernel-level techniques to application designers, which can use IMs to selectively create, update and recover critical data copies. Therefore, applications such as the transactional service where only some pages are modified between updates and where critical data is only a part of the process' address space, should show significant performance gains when using IMs. Still, IMs are not well suited to applications which modify most of their critical data pages between messages, and which should better use other IPC mechanisms.

The experiments reported in this section have been performed with processes residing on the same node, a PC with an Intel 486DX2-66 with 16MBytes of RAM and 216MB of swap on an IDE Hard Disk, with 1.92 ms as average seek time, and 1210 KB/s of transfer rate.

5.1. IM Analysis

This experiment has been performed with two processes. The main one holds the working copy of the data, and performs controlled writes on randomly selected pages. The second is an R process as described in the previous section that just waits for IMs to arrive and then processes them immediately. For each size, 50 IM were sent. Between two messages, the main process modifies the specified number of randomly selected pages. Figure 7 shows the mean IM transfer time of IMs versus the size of the memory region. For sizes smaller than 3000 pages, the cost strongly increases as the size decreases. The unexpected behavior (we expected a small increment as the size increased) is caused by blocking due to synchronization among IM services and the page stealing process.[3] When the size of the region is small, blocking is more frequent. On the other hand, from 3000 pages onwards, the cost increase is due mainly to the fact that as the region grows there are more page misses.

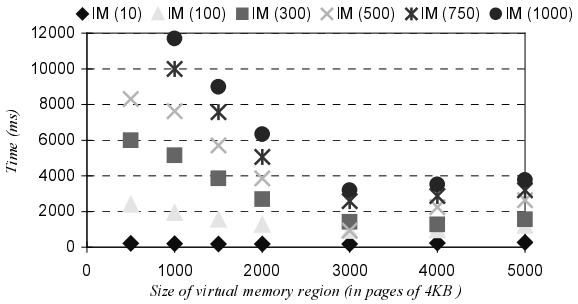

Figure 7: Temporal cost of IMs versus size of VM regions for 10, 100, 300, 500, 750 and 1000 modified pages.

This experiment shows the relevance of synchronization delays, and suggests that IMs should be considered as an integral part of the virtual memory management. For example, in our implementation, a page remains locked while it is being filled or written back to disk. This means that the IM send routine should block if it needs to access the modified page bitmap (see subsection 3.3). However, if a richer blocking mechanism were used, disk transfers would not need to block IM routines, since they (disk transfers) do not alter the modified bitmap for the page.

5.2. Checkpointing Analysis

The overhead that checkpointing implemented using IMs produces on an intensive computing application has been measured on a program that multiplies 615x615-integer matrices. Both matrices reside on a file; the program reads them, performs the product and then writes the result back to disk. Six different checkpointing techniques have been analyzed (see Table 3), which have been implemented by library code that uses IMs. Table 4 shows the results of these tests. Checkpoints have been established 11 times during the execution, which means one every 50 s.

Table 3: Checkpointing techniques used.

N.	*Description*
0	No checkpointing.
1	Incremental to memory. An IM is sent to the back-up R process.
2	Incremental to disk, synchronous. An IM is sent to a back-up process, which receives it packed and uses the map to update a back-up file. Then it acknowledges the sender.
3	Incremental to disk, asynchronous. As in the previous one, but the acknowledgment is sent before the file is updated.
4	Packed to disk, synchronous. As in method 2, but writing the raw PIM to a new file.
5	Packed to disk, asynchronous. As the previous one, but async.
6	Non incremental to disk. The whole result matrix is written to a file.

Table 4: Temporal cost of checkpointing (CP) using IMs.
"run. t" stands for total execution time, "ov" for overhead,
"t" for mean time to establish one CP and "sz" for CP size.

Method	run. t (s)	ov (s)	ov (%)	t (s)	sz (MB)
0	547,9				
1	548,6	0,7	0,1	0,03	0,39
2	566,9	19,1	3,5	1,71	0,39
3	566,6	18,8	3,4	0,05	0,39
4	557,3	9,4	1,7	0,73	0,39
5	556,3	8,5	1,5	0,02	0,39
6	572,3	24,4	4,4	2,04	1,71

It can be noted how, compared to method 6, IMs significantly reduce the overhead due to checkpointing, and this reduction is more important when PIMs are used. It should be noted the high performance that can be obtained when the back-up copy remains in memory (method 1). It is also worth to stand out that methods 2 and 3 incur in an overhead similar to that of method 6. This is caused by the high number of messages exchanged as every modified page is sent to disk in a separate write operation (each one generates one message from process R to the file server, one from the file server to the disk server, and the necessary replies).

These results are similar to those reported in [6] for the same application when user-directed checkpointing is used. Nevertheless, both experiments are not directly comparable since there are significant differences between them, such as the operating system structure (Unix versus VSTa, which is a micro-kernel), or the place where checkpoints are saved (NFS imported file system versus in-memory or local file system).

5.3. Transactional File System

To show how incremental operations are easy to use for implementing fault tolerant applications, a small transactional file server has been implemented. The user has access from the command line to three basic operations, StartT, EndT and AbortT, which cause the client process to invoke these operations on the server. With this structure, off-the-shelf utilities (such as text editors, compilers, or file management utilities) can benefit from the fact of being used inside transactions, and thus their effects be discarded or confirmed as required.

[3] Note that even when only 100 pages are modified between two consecutive IMs, as pages are selected randomly and 50 IMs are sent for each region size, page stealing and page fault treatment are intensive, because with 16MB of RAM there are only 4096 pages available, and they must be shared with the OS kernel and system servers.

The transactional file server has been implemented following the concurrent transactional service described in subsection 4.3. This structure requires: (1) that data are stored in one virtual memory region, and (2) that the smallest unit to be assigned to a file is a page (because concurrency is managed at page level).

A new file system, called *cfs*, which satisfies these requirements, has been implemented on VSTa [20]. Measures have been carried out with a storage region of 5000 pages of 4 KB. Table 5 shows the results of performance comparison among the transactional file server (*ctfs*), the file system implemented on VSTa (*cfs*), the standard temporal file server of VSTa (*tmpfs*) and the FAT (MS-DOS) file system of VSTa. As the table shows, *ctfs* offers similar performance to *cfs* and *tmpfs*, which are memory-based, and much better performance than FAT, which is disk-based.

Table 5: Comparison of "ctfs" versus "cfs", "tmpfs" and "FAT".
(Times expressed in ms.)

Operation	Size	ctfs	cfs	tmpfs	FAT
Read	512 B	1,1	1,6	1,6	26,8
	64 KB	48	60,6	59,4	105
	1 MB	736	967	949	1977
Write	512 B	0,89	2,0	2,1	28,5
	64 KB	38,8	64,2	60,3	57,7
	1 MB	616	986	912	1853

6. Conclusions and Future Work

This paper proposes Incremental Messages (IM), a new type of kernel services designed to efficiently and flexibly manage replicated copies of critical data. These services combine the advantages of both kernel and user-level approaches to checkpointing and recovery. As kernel mechanisms do, IMs take advantage of having direct access to the memory management unit, providing efficient transfer of data by means of concurrent and incremental copying. At the same time, the proposed interface is flexible enough to let user-level applications establish which data must be copied and when, permitting that only critical data be copied.

As section 4 describes, passive replicates, user-level checkpointing and transactional services can be implemented easily and efficiently using IMs. Furthermore, error detection and confinement can also be increased by fully exploiting the proposed interface. Hence, IMs are a valuable tool to support the development of fault-tolerant applications, justifying the cost of including them in a microkernel.

The first experimental results obtained following the implementation of IMs on VSTa suggest that the current IM implementation is efficient enough to build practical fault-tolerant applications. However, they also show that the efficiency of these services strongly depends on the design of the virtual memory subsystem and, specially, on the paging strategy. Consequently, to fully exploit the performance improvement that IMs should offer, they should be accounted for as an integral part of the virtual memory management design. This is a limitation of the current implementation, which should be addressed in future works.

References

[1] J.S. Plank, "Efficient checkpointing on MIMD architectures", *PhD Thesis*, Dep. of CS, University of Princeton, June 1993.

[2] E.N. Elnozahy, "Manetho: Fault-Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication", *PhD Thesis*, Rice University, October 1993.

[3] F. Cristian, B. Dancey, and J. Dehn, "Fault-Tolerance in the Advanced Automation System", *In Proc. of the 20th FTCS*, June 1990, pp. 6-17.

[4] Y. Huang, and C. Kintala, "Software Implemented Fault Tolerance: Technologies and Experience", *In Proc. of the 23rd FTCS*, June 1993, pp. 2-9.

[5] D. Powell, "Distributed Fault Tolerance: Lessons from Delta-4", *IEEE Micro*, February 1994, pp. 36-47.

[6] J. S. Plank, M. Beck, et al., "Libckpt: Transparent Checkpointing under Unix", *USENIX Winter 1995 Technical Conference*, January 1995, pp. 213-223.

[7] C. Pérez, G. Fabregat, et al., "Micro-Kernel support for Fault-Tolerant Application Development on Distributed Systems", *In Digest of FastAbstracts: FTCS-28*, IEEE CS, June 1998, pp. 43-44.

[8] V. Abrossimov, F. Herrmann, et al., "Fast Error Recovery in CHORUS/OS: The Hot-Restart Technology", *CSI-T4-96-34 Technical Report, Chorus Systems, Inc.*, August 1996.

[9] J. Lipkis, and M. Rozier, "Fault Tolerance Enablers in the CHORUS Microkernel", *Hardware and Software Architectures for Fault Tolerance*, Lecture Notes in Computer Science nº 774, Springer-Verlag, 1994, pp. 182-190.

[10] M. Rozier, V. Abrossimov, et al., "Overview of the Chorus Distributed Operating System", *USENIX Micro-kernels and Other Kernel Architectures Workshop Proc.*, April 1992, pp. 39-69.

[11] R. van Renesse, and A.S. Tanenbaum, "Short Overview of Amoeba", *USENIX Micro-kernels and Other Kernel Architectures Workshop Proc.*, April 1992, pp. 1-10.

[12] D.L. Black, D.B. Golub, et al., "Microkernel Operating System Architecture and Mach", *USENIX Micro-kernels and Other Kernel Architectures Workshop Proc.*, April 1992, pp. 11-30.

[13] A. Valencia, "An Overview of the VSTa Microkernel", URL:*http://bodhi.zendo.com/vsta/vsta_intro.html*.

[14] S. Levi, and A.K. Agrawala, "Fault Tolerant System Design", *McGraw-Hill*, 1994.

[15] J.-C. Laprie, J. Arlat, et al., "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures", *Predictably Dependable Computing Systems*, Basic Research Series, Springer, 1995, pp. 103-122.

[16] G. Deconinck, J. Vounckx, et al., "Survey of Checkpointing and Rollback Techniques", *Project ESPRIT 6731 (FTMPS) Technical Report*, June 1993.

[17] G. Coulouris, J. Dollimore, and T. Kindberg, "Distributed Systems. Concepts and Design", *Addison-Wesley,* 2nd edition, 1996.

[18] R. Draves, "A Revised IPC Interface", *USENIX Mach Workshop Proc.*, October 1990, pp. 101-122.

[19] H. Madeira, and J.G. Silva, "Experimental Evaluation of the Fail-Silent Behavior in Computers Without Error Masking", *In Proc. of the 24th FTCS*, June 1994, pp.350-359.

[20] C. Pérez, "Aportaciones a los Entornos de Desarrollo de Aplicaciones Tolerantes a Fallos", *PhD Thesis*, Dpto. de Informática y Electrónica, Universitat de València, February 1998.