Failfast creates a need for instant crash and restart. This again may seem a preposterous approach, but computer system architectures are increasingly adopting this approach. The concept of *process pair* (covered in Subsection 3.7.3) specifies that one process should instantly (in milliseconds) take over for the other in case the primary process fails. In the current discussion, we take the more Olympian view of *system pairs,* that is two identical systems in two different places. The second system has all the data of the first and is receiving all the updates from the first. Figure 3.2 has an example of such a system pair. If one system fails, the other can take over almost instantly (within a second). If the primary crashes, a client who sent a request to the primary will get a response from the backup a second later. Customers who own such system pairs crash a node once a month just as a test to make sure that everything is working—and it usually is.

If Heisenbugs are the dominant form of software faults, then failfast plus transactions plus system pairs result in software fault tolerance. Geographically remote system pairs tolerate not just Heisenbugs, but many other problems as well. They tolerate environmental faults, operator faults, maintenance faults, and hardware faults. Two systems in two different places are not likely to have the same environmental problems: They are on different power grids, different phone grids, and different earthquake faults, and they have different weather systems overhead. They have independent operations staffs, different maintenance personnel, and different hardware. All this means that the two systems have largely independent failure modes. Section 12.6 develops the concept of system pairs in more detail.

3.6.3 Summary

Software faults are the dominant source of system failures. All other faults can be masked with a combination of redundancy, geographic diversity, and software to automate tasks. Software can automate system operations and mask operations and maintenance failures. Software faults, however, remain an unsolved problem.

There are two approaches to software fault tolerance: *n*-version programming and transactions. The two approaches could be combined. Advocates of *n*-version programming aim to combine several incorrect programs into a better, more reliable one. *N*-version programming may also be a good way to write failfast programs. Transaction advocates aim to detect incorrect programs and minimize their effects by undoing them. Transactions encourage a failfast design by allowing the system to quickly crash and restart in the most recent consistent state. By having a standby system, restart can begin within milliseconds.

3.7 Fault Model and Software Fault Masking

The application of the pair-and-spare or *n*-plex techniques to software modules is not obvious. How do you pair-and-spare a software module? How do you *n*-plex messages, remote procedure calls, and the like? The answer to these questions—process pairs—is neither trivial nor a direct application of the hardware *n*-plex and pair-and-spare approaches described in the previous section. To the authors' knowledge, the best approach to process pairs was worked out in an unpublished classic written by Butler Lampson and Howard Sturgis in 1976 at Xerox. At about the same time Joel Bartlett designed and implemented similar ideas for Tandem's Guardian operating system. The Lampson-Sturgis model has widely influenced subsequent work in the field; but, unfortunately it is not widely available.

Bartlett's work is even more inaccessible. The presentation here borrows heavily from those original works. The process pair discussion comes from experience with Bartlett's design.

Designing fault-tolerant programs requires a model. The model must define correct behavior, and if the programs are to deal with faults, the model must describe the kinds of faults and their relative frequencies. Given such a model, programs can be written, system reliability can be estimated using probabilistic methods, and proofs that the programs are correct can be made with any desired degree of formality. The validity of the model cannot be established by proof, since there is no formal model of reality. The strongest possible claim is that the physical system behaves like the model with probability p, and that p is very close to 1. This claim can only be established empirically, by estimating p from measurements of real systems.

The model here involves three entity types: *processes, messages,* and *storage.*[5] Each has a set of desired behaviors and a set of failure behaviors. This section shows how to transform each of the three entities from failfast entities into highly reliable and even highly available entities.

Faulty behavior is dichotomized into *expected* faults (those tolerated by the design), and *unexpected* faults (those that are not tolerated). Unexpected faults can be characterized as dense faults or Byzantine faults:

Dense faults. The algorithms will be n-fault tolerant. If there are more than n faults within a repair period, the service may be interrupted.

Byzantine faults. The fault model postulates certain behavior—for example it may postulate that programs are failfast. Faults in which the system does not conform to the model behavior are called Byzantine.

It is highly desirable that the system be failfast in the dense fault cases. That is, if there are more than n-faults in the repair window, the system should stop rather than continue the computation in some unspecified way. It is possible to measure the rate of unexpected faults in real systems and observe that the rate is small. Some representative measurements appeared in Sections 3.3 and 3.4.

Picking the model is a very painful and delicate process. One is constantly torn between simplicity and completeness. For example, when a storage module breaks and is repaired, in our model it returns to service in the empty state. A more complete model would have two kinds of repair. But a more complete model would be much bigger and would not make the ideas much clearer. The goal here is to provide a basis for understanding the fundamental issues and algorithms.

3.7.1 An Overview of the Model

The model describes each aspect of the system—storage, processors, and communications—by presenting a program that simulates the behavior of such entities, complete with their failure characteristics. For example, a storage module reads and writes data at storage addresses, but it also occasionally writes the data to the wrong place, invalidates a page, or

[5] It would be possible to dispense with storage, since storage modules are just processes. This is the view of the actor's model [Agha 1986]. No one has been able to unify the concepts of process and messages. Perhaps, by analogy with energy and matter, the two concepts are interchangeable, but both are needed because their functions are so different.

damages the whole store (the latter two events require the store to be repaired). Given these errant storage modules, the presentation then shows programs to mask the failures through failvote duplexing and writes a program to repair the failures in the background.

The presentation then models processes and messages. Messages can be lost, duplicated, delayed, corrupted, and permuted. By implementing sessions, timeouts, and message sequence numbers, all message failures are converted to lost messages. By combining this simple failure model with message acknowledgment and sender timeout plus message retransmission, the message system becomes highly available.

These techniques for building reliable stores and reliable messages demonstrate how software masks hardware faults, but they have little to do with masking software faults. Still, the presentation is instructive in that it sets the stage for the software fault tolerance discussion that follows.

Processes fail by occasionally being delayed for some repair period, having all their data reset to the null state, and then having all their input and output messages discarded. The presentation then uses this model to show how to build process-pairs. One process, called the primary, does all the work until it fails; then the second process, called the backup, takes over for the primary and continues the computation. During normal processing, the primary periodically sends *I'm Alive* messages to the backup. If the backup does not receive an *I'm Alive* message from the primary for a couple of messages periods, it assumes the primary has failed and takes over for the primary. Three kinds of takeover are described:

Checkpoint-restart. The primary records its state on a duplexed storage module. At takeover, the backup starts by reading these duplexed storage pages.

Checkpoint message. The primary sends its state changes as messages to the backup. At takeover, the backup gets its current state from the most recent message.

Persistent. The backup restarts in the null state and lets the transaction mechanism clean up (undo) any recent uncommitted state changes.

The benefits and programming styles of these three forms of process pairing are contrasted. To preview that discussion:

Quick repair. Process pairs obtain high availability processes by providing quick process repair.

Basic Pairs must Checkpoint. Certain programs are below the transaction level of abstraction and must therefore use some form of checkpointing to get highly available program execution. Examples of such primitive programs are the transaction mechanism itself, the operating system kernel, and the programs that control physical devices (disks and communications lines).

Persistent is Simple. Checkpointing of any sort is difficult to understand. Most people will want to use transactional persistent processes instead.

Process pairs mask hardware failures (processor failures) as well as transient software failures (Heisenbugs). As such, they are the key to software fault tolerance.

This concludes the overview. The following discussion turns to describing the behavior of storage, processes, and messages by the behavior of programs that simulate their

behavior. The Lampson-Sturgis model is very instructive and provides a clear way of contrasting some subtle issues. However, the material is very challenging. Readers who do not care to delve in depth into this subject may wish to skim the rest of this section.

3.7.2 Building Highly Available Storage

Reliable storage is built as follows: First, the basic storage operations and failure behavior are defined. Then higher-level operations are defined; with high probability, these mask the failure behavior by *n*-plexing the devices. This is the analog of the hardware discussion of Section 3.6, but it is less abstract, since specific modules and specific failure modes are involved.

3.7.2.1 The Structure of Storage Modules

A storage module contains an array of *pages* and a *status* flag. If the module status is FALSE, then it has failed, and all operations on it return FALSE. Each page of a store has an *address*, a *value*, and a *status*. Addresses are positive integers up to some limit. The status is either TRUE or FALSE. If the page status is FALSE, the page value is invalid; otherwise, the page value stores the data most recently written to it. Two operations are defined on storage modules—one to write page values, and another to read them. Rewriting a page makes it valid (makes its status TRUE). Intuitively, these definitions are designed to model disks or RAM disks.

3.7.2.2 Definition of Store and of Store Read and Write

The definitions of the data structures for the programs are given in the following listing and are illustrated in Figure 3.17.

```
#define  MANY          100000        /*a big number (system will have many entities)
#define  MAXSTORES     MANY          /*with many stores
#define  MAXSTORE      MANY          /* each holding many pages.
#define  VSIZE         8192          /* size of a page value (also used for messages)
typedef  char *        avalue;       /* each value will be 8kB
typedef  Ulong         address;      /* address is integer 0≤x<MAXSTORE
typedef  struct {                    /* template for a page
              Boolean  status;       /* status is true if page is ok
              char     value[VSIZE]; /* value of the page
              } apage;               /*
typedef  struct{                     /* define the template for a storage module
              Boolean  status;       /* flag indicates that module is functioning
              apage    page[MAXSTORE]; /* storage module has many pages
              } astore;              /*
astore   stores[MAXSTORES];          /* allocates the array of many stores
```

Storage objects support two operations: read and write. They are defined by the simple code fragments that follow. Notice in particular that writes may occasionally fail (with probability pwf) by having no effect.

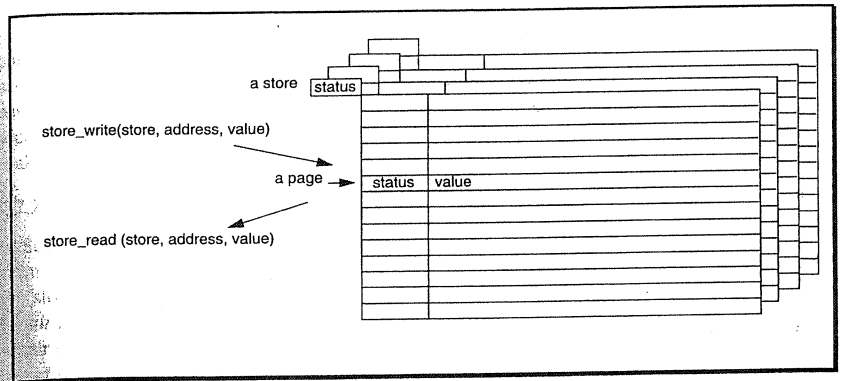

Figure 3.17: The data structures and verbs used by the model of storage. There are many stores, each of which carries a status flag indicating it is valid. Each store consists of many pages; each page has a status flag and a value. Such stores model disks and RAM disks.

```
#define     pwf     1E-6                 /*probability write has no effect (1 in a million)  */
float       randf();                     /* returns a random float 0<x<1 (never 0 or 1)       */
/* write a value to a particular page of a particular store.                                  */
Boolean store_write(astore store, address addr, avalue value) /*                            */
    { if (addr >= MAXSTORE) return FALSE;   /* return false if bad address                */
    if (!store.status) return FALSE;        /* return false if store has faulted          */
    if (pwf > randf()) return TRUE;         /* sometimes write faults by doing nothing    */
    store.page[addr].status = TRUE;         /* set page status to valid                   */
    copy(store.page[addr].value,value, VSIZE); /* else store.page[addr].value=value       */
    return TRUE;                            /*                                            */
    };                                      /*                                            */
Boolean store_read(astore store, address addr, avalue value)  /*                          */
    { if (addr >= MAXSTORE) return FALSE;   /* return false if bad address                */
    if (!store.status) return FALSE;        /* return false if store has faulted          */
    if (!store.page[addr].status) return FALSE; /* or bad page value                      */
    copy(value,store.page[addr].value, VSIZE); /* else value=store.page[addr].value       */
    return TRUE;                            /*                                            */
    };                                      /*                                            */
```

These programs model, or simulate, durable storage devices such as disks, tapes, and battery-protected electronic RAM. For simplicity, the code does not model soft read faults (a read that fails but will be successful on retry). It does model the simple case of a write occasionally having no effect at all (null writes in the third statement of store_write()). This happens rarely, with a probability pwf (probability of write failure). As explained in the next paragraph, address faults (a write of a page other than the intended page) are modeled by page decay (spontaneous page failure) and by null writes (writes that have no effect).

The model assumes that reads and writes of incorrect locations produce a FALSE status flag and consequently are detectable errors. Both appear to be a spontaneous decay of the

page (see below). Incorrect reads and writes are typically detected by storing the page address as part of the page value, so that a read of a wrong page produces a FALSE status flag. This mechanism also detects a write of a correct value to an incorrect address when the overwritten data is subsequently read, because the page address stored in the value will not match the page's store address. In addition, each page is covered by a checksum that is used as follows:

Write. When a page is written, the writer computes the page checksum and stores it in the page, and

Read. When the page is read, the checksum is recomputed from the value and compared to the original checksum stored in the page.

If the checksums do not match, the page is invalid. All this validity checking is implicit in the status flag.

3.7.2.3 Storage Decay

Each store and each page may fail spontaneously, or a page may fail due to an incorrect store_write(). The spontaneous failure is modeled by a *decay process* for each store. Operating in the background, this decay process occasionally causes a page to become invalid (status = FALSE) or even invalidates the entire store. The model postulates that store and page errors are independent and that the frequency of such faults obeys a negative exponential distribution with means MTTVF and MTTSF, respectively. Table 3.10 suggests values for such means. The storage decay process is given in the code segment that follows.

```
/* There is one store_decay process for each store in the system
#define  mttvf    7E5                    /* mean time (sec) to a page fail, a few days
#define  mttsf    1E8                    /* mean time(sec) to disk fail is a few years
void store_decay(astore store)           /*
   { Ulong  addr;                        /* the random places that will decay
   Ulong page_fail = time() - log (mttvf*randf());   /* time in seconds to next page decay
   Ulong store_fail = time() - log (mttsf*randf());  /* time in seconds to next store decay
   while (TRUE)                          /* repeat this loop forever
      { wait(min(page_fail,store_fail) - time());    /* wait for next event
      if (time() >= page_fail)           /* if the event is a page decay
         { addr = randf()*MAXSTORE;      /* pick a random address
         store.page[addr].status = FALSE;   /* set it invalid
         page_fail = time() - log(randf())*mttvf;   /* pick a wait time for next fault
         };                              /* negative exp distributed, mean mttvf
       if (time() >= store_fail)         /* if the event is a storage fault
         { store.status = FALSE;         /* mark the store as broken
         for (addr = 0; addr < MAXSTORE; addr++)    /*invalidate all page values
            store.page[addr].status = FALSE;        /*
         store_fail = time() - log(randf())*mttsf;  /* pick a time for next fault
         };                              /* negative exp distributed, mean mttsf
      };                                 /* end of endless while loop
   };                                    /*
*/
*/
*/
*/
*/
*/
*/
*/
*/
*/
*/
*/
*/
*/
*/
*/
*/
*/
*/
*/
*/
```


As defined, storage gradually decays completely. According to the parameters described earlier, half the stores will have decayed after three years. Such stores are not reliable—reliable devices continue service from their pre-fault state when the fault is repaired. The model assumes that each store and each store page are failfast and are repaired and returned to service in an empty state; that is, all pages have status = FALSE. Algorithms presented next will make them reliable.

3.7.2.4 Reliable Storage via *N*-Plexing and Repair

In Section 3.6, we saw that repair is critical to availability. The goal is to build reliable stores from off-the-shelf, unreliable components. A reliable store can be constructed by *n*-plexing the stores, by reading and writing all members of an *n*-plex group, and by adding a *page repair process* for each group of stores. Storage is assumed to have exponentially distributed repair times with a mean of a few hours (MTSR $\approx 10^4$ seconds). This includes the latency to detect storage faults.

The idea here is to divide the stores into groups of *n* members and then to pretend that there are only

$$\frac{MAXSTORES}{n}$$

logical stores. Each reliable_write() attempts to write all members of the group, and each reliable_read() attempts to read all members of the group.[6] The write case is easy and is represented by the following code:

```
#define  nplex    2                     /* code works for n>2, but do duplex   */
Boolean reliable_write(Ulong group, address addr, avalue value) /*            */
   { Ulong      i;                      /* index on elements of store group    */
   Boolean status = FALSE;              /* status is true if any write worked  */
   if (group >= MAXSTORES/nplex) return FALSE;   /* groups = 1/nth of stores   */
   for (i = 0; i < nplex; i++ )         /* write each of the n stores in the group. */
      status = status ||                /* status indicates if any write worked */
            store_write(stores[group*nplex+i],addr,value);   /*                 */
   return status;                       /* and return status                   */
   };                                   /*                                     */
```

Note that a really careful design might read the value after writing to make sure that at least one of the pages now has the correct value.

The reliable_read() faces two problems:

All fail. If none of *n* members can be read, then reliable_read() fails. This should be rare. If its frequency is unacceptably high, then *n* can be increased.

Ambiguity. If *some* of the *n* members are readable (all have good status) but give different values, there is a dilemma: which value should the reliable_read() return? The problem can easily arise if some earlier writes fail (had no effect or had an address fault). This problem can't be solved by increasing *n*; in fact the frequency of the problem increases with *n*. Setting *n* to 1 "solves" the problem but does not give a reliable store.

---

[6] Concurrently writing *n* replicas creates synchronization problems if multiple writers race multiple readers and writers. For the moment, assume that each operation is serialized as a unit. Serializing the operations is covered later.

One solution to resolving ambiguous reads is to take the majority view, the most popular value. But that only works if there is a majority value. If there are two stores and two values (the most common case), there is no majority.

The premise is that there is a correct value—the most recently written one.[7] Thus it is postulated that each page has a *version* generated at the time of the write as part of the value.[8] The key property is that version numbers increase monotonically. Page value versions are assumed to have the property

(version(value1) > version(value2)) *if and only if* (value1 was written more recently than value2).

If the *n*-plex reliable_read() discovers a version mismatch, it takes the most recent value and writes that to all other members of the group. In addition, if it finds a member with bad status, it rewrites that member with the most recent version. This could be left to the repair process, but it is repeated here to minimize the latency between fault detection and repair. The reliable read code for *n*-plex stores follows.

```
Ulong version(avalue);                                  /* returns version of a value
/* read an n-plex group to find the most recent version of a page
Boolean reliable_read(Ulong group, address addr, avalue value) /*
                                                        /* index on store group
    { Ulong     i       = 0;                            /* flag says had a good read
    Boolean     gotone  = FALSE;                        /* bad says group needs repair
    Boolean     bad     = FALSE;                        /* next value that is read
    avalue      next;                                   /* read ok
    Boolean     status;                                 /* groups = 1/nth of stores
    if (group >= MAXSTORES/nplex) return FALSE;         /* return false if bad address
    if (addr >= MAXSTORE) return FALSE;                 /* for each page in the nplex set
    for (i = 0; i < nplex; i++ )
        { status = store_read(stores[group*nplex+i],addr,next); /* read its value
        if (! status ) bad = TRUE;                      /* if status bad, ignore value
        else                                            /* have a good read
            if (! gotone)                               /* if it is first good value
                {copy(value,next,VSIZE); gotone = TRUE;}/* make it best value
            else if ( version(next) != version(value))  /* if a second value, compare
                { bad = TRUE;                           /* if different, repair needed
                if (version(next) > version(value))     /* if new value is best version
                copy(value, next, VSIZE);               /* copy it to best value
        };          };                                  /* end of read all copies
    if (! gotone) return FALSE;                         /* disaster, no good pages
    if (bad) reliable_write(group,addr,value);          /* repair if saw any bad pages
    return TRUE;                                        /* success
    };                                                  /*
```

[7] This is a major difference between the software view of redundancy and the hardware view. The hardware view must insist on a majority, because it has no understanding of the data being stored. The software often has very clear ideas about the meaning of the data.

[8] In Subsection 10.3.7 this version is the log sequence number (called the page LSN), but it could just as well be a timestamp reliably generated by the hardware.

If an *n*-plexed page is not read for many years, all *n* copies of it will likely decay. To prevent this scenario, a store repair process runs in the background for each *n*-plex group. This process merely reads each page of the group once a day (or at some other frequency). The reliable_read() routine performs any needed repair. As part of its normal operation, it reads all copies and updates them if any disagree or are bad. The store repair process repairs any broken pages in a store within a day of its failure, since it visits each page about that often.

```
/* repair the broken pages in an n-plex group. Group is in 0,...,(MAXSTORE/nplex)-1  */
void store_repair(Ulong group)                          /*                           */
    { int       i;                                      /* next address to be repaired */
    avalue  value;                                      /* buffer to hold value to be read */
    while (TRUE)                                        /* do forever                */
        for (i = 0; i <MAXSTORE; i++)                   /* for each page in the store */
            { wait(1);                                  /* wait a second             */
            reliable_read(group,i,value);               /* do a reliable read (that will repair pages */
            };                                          /*                   if they do not match) */
```

3.7.2.5 Optimistic Reads of the *N*-Plex Group

N-plexing of disks and other storage is common and is implemented as described previously. Writes are done to both disks. But often, especially in the duplexed disk case, the read operation is *optimistic*: it is directed to only one disk. If that fails, then the other disk is consulted. The logic is something like this:

```
Boolean optimistic_read(Ulong group,address addr,avalue value) /*
    {if (group >= MAXSTORES/nplex) return FALSE;    /* return false if bad disk address */
    if (addr >= MAXSTORE) return FALSE;             /* return false if bad page address */
    if (store_read(stores[nplex*group],addr,value))  /* read one value and if that is ok */
        return TRUE;                                /* return it as the true value      */
    else                                            /* if reading one value returned bad */
        return (reliable_read(group,addr,value));   /* then do n-plex read and repair.  */
    };                                              /*                                  */
```

Optimistic reading is dangerous. Suppose one of the *n* store_write() operations within a reliable_store_write() fails; that is, suppose it is a null operation. Then, that storage group of pages will have different page values and different page versions. If only one member of the *n*-plex is read, it may have the old version rather than the new version, a *stale read*. Stale reads occur half the time for a duplexed disk page damaged by a failed write. Believers in optimistic reads assume each disk is failfast. If so, the writer can sense the write failure and immediately initiate recovery. Recovery from the write failure includes retrying the write, sparing the page to a new location (remapping the address to a new storage page), or ultimately invalidating the whole storage module containing the bad page. The stale write problem often is not carefully thought through, and it is a typical flaw in optimistic *n*-plex schemes. To summarize, optimistic reads get the performance benefits of reduced read traffic but risk reading stale data.

The implementation of storage operations assumes that each *n*-plexed read and write is atomic. That is, if I do a read while you are doing a write, then I will see the store as it was

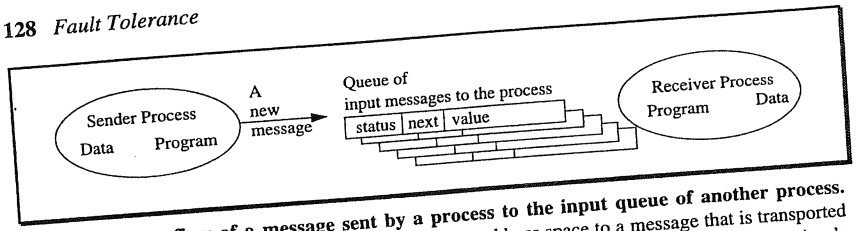

Figure 3.18: The flow of a message sent by a process to the input queue of another process. Message_send() operation copies data from the sender's address space to a message that is transported to the receiver. When the receiver process executes message_get(), the data is copied to the receiver's address space. A status flag indicates whether the message has been corrupted in transit. When a process fails, its data and input message queue are reset to their initial (null) state.

before or after your write, but not the state during your write. A similar assumption applies to two concurrent write operations. The usual way to ensure this one-at-a-time serialization is to have some (reliable and available) process perform the storage reads and writes for everyone else. Such a server, which serializes the requests, could be appointed for each *n*-plex page or for any larger group. Alternatively, a locking scheme can be used. A highly available and reliable server manages the lock requests for individual pages and serializes access to the pages, but it actually does none of the work. Each of these designs suggests the need for reliable processes, which is the next topic.

The preceding discussion covered *n*-plexing of storage. A different approach is to use one or more stores to hold an error-correcting code for a set of stores [Patterson et al. 1988]. This approach could also be described and analyzed using the model presented here. That is left as a term-project exercise.

3.7.2.6 Summary

A simple model of storage and storage failures captures much of the logic used in I/O subsystems. It allows careful definition of *n*-plexed storage modules and illustrates the version mismatch problem for *n*-plex stores as well as the stale-version problem for optimistic reads. In addition, it points out the need for the store_repair() process and the simplicity of that process. Given the statistics used by this model (see Sections 3.3 and 3.4), a computation of the probability of various failure modes shows that duplexing modern disks and other nonvolatile storage devices provides highly reliable and highly available storage.

3.7.3 Highly Available Processes

Highly reliable and available processes can be constructed from failfast processes and messages. Processes and their interactions cannot be discussed without talking about messages; unfortunately, the opposite is also true. Because we have to begin somewhere, we start by defining messages. The treatment of messages beyond their definition is reserved for the next subsection. Indeed, it will turn out that reliable messages require reliable processes to implement them.

3.7.3.1 Messages Defined

Processes interact by sending messages to one another or by writing to shared storage that is later read by others. A *message* is a value sent from one process to another. Figure 3.18 diagrams the flow of a message from a sending process to a receiving process.

Message delivery is not reliable. Some messages are lost (they are never delivered). Some may be delayed, arbitrarily permuting the message delivery order to a process's input queue. Even if a message is delivered, there is some small chance it will be *corrupted*, so that the message data is unreadable. The loss and corruption of messages is quantified as the probability of message failure, pmf. There is a comparable chance that the message will be *duplicated*, so that the message appears twice in the process's input queue. The following example code demonstrates the external definition of message gets and sends.

```
/* definitions of constants and data structures used by the message simulation                */
#define pmf      1E-5                  /* probability of message failure (loss or corrupt)   */
#define pmd      1E-5                  /* probability of message duplication                 */
typedef struct Message * amessagep;    /* pointer to a message structure                     */
typedef struct Message{                /* struct defining message formats                    */
        Boolean      status;           /* status is true if message data (value) is ok       */
        amessagep    next;             /* pointer to next input message of this process      */
        avalue       value;            /* data in the message                                */
        } amessage;                    /*                                                    */

/* Send a message to a process, returns true if process exists                                */
Boolean message_send(processid him, avalue value);  /*                                       */
/* Get the next input message of this process: Returns true if there is a message else false. */
/*                                  In true case, valuep addresses message, status is msg status */
Boolean message_get(avalue * valuep, Boolean * msg_status);    /*                             */
```

The next piece of code gives a sense of how these routines are used, and it defines a useful routine. This get_msg() routine discards corrupt input messages and returns false if there are no good new messages in the process input queue, or returns true and the data of the next input message.

```
/* return pointer to next valid input message in valuep and return TRUE or return FALSE      */
Boolean get_msg(avalue * valuep)                    /*                                       */
  { Boolean    msg_status = FALSE;                  /* flag says msg value is not corrupt    */
    Boolean    msg_exists = FALSE;                  /* flag says next input msg exists       */
    while (msg_status = FALSE)                      /* read until have good msg or queue empty */
        {   msg_exists = message_get(&valuep, &msg_status); /* get next msg if it exists     */
            if (!msg_exists) break;}                /* if input queue is empty, quit read loop */
    return msg_exists;                              /* return msg to caller                  */
    };                                              /*                                       */
```

This completes our brief overview of messages. Messages are explored further in Subsection 3.7.4, following the introduction of reliable processes.

3.7.3.2 Process Fault Model

A process executes a program, transforms the process state, and displays an execution behavior. The process's *external behavior* is a sequence of store_read(), store_write(), message_send(), and message_get() operations. This external behavior is created by the process

executing its program logic on the process state. Each process state is a bundle of bytes containing its program and data. Each process has two states, an *initial state* that never changes and a *current state* that changes at each program step. In addition, each process has an *input queue*, an unordered set of messages sent to it by other processes.

```
#define  MAXPROCESS  MANY                    /* the system will have many processes
typedef Ulong processid;                     /* process id is an integer index into array
typedef struct {char program[MANY/2];char data[MANY/2]} state; /* state is program + data
struct { state          initial;             /* process initial state
         state          current;             /* value of the process state
         amessagep      messages;            /* queue of messages waiting for process
         } process [MAXPROCESS];              /*
```

A process fails by stopping for a while (say, an hour) to be repaired, then being reset to its initial state, and having all its unprocessed input messages discarded. That is analogous to a restart of processors in many operating systems. At any instant, each process runs the risk of faulting. After a reset, the process executes its restart logic and continues processing. Of course, it runs the risk of experiencing another reset during the restart.

Here is a way to think of process faults: Imagine that each process is implemented by a lower-level automaton that usually executes the next step of the process, but sometimes stops the process execution for a repair period, discards all the process' messages, resets the process to the initial state, and then continues the process execution from the initial state (this state includes the process' program). In addition, the process may have defensive programming tests that cause it to failfast by calling panic() if the test is false. These calls to panic are modeled as just another aspect of the execution automaton occasionally failing the process. Such an automaton is defined as follows:

```
/* Process Decay : execute a process and occasionally inject faults into it
#define  mttpf    1E7                        /* mean time to process failure ≈4 months
#define  mttpr    1E4                        /* mean time to repair is 3 hours
void process_execution(processid pid)        /*
    { Ulong       proc_fail;                 /* time of next process fault
      Ulong       proc_repair;               /* time to repair process
      amessagep   msg, next;                 /* pointers to process messages
      while (TRUE)                           /* global execution loop
          { proc_fail = time() - log(randf())*mttpf;   /* the time of next fail
            proc_repair = -log(randf())*mttpr;         /* delay in next process repair
            while (time() < proc_fail)                 /*
                { execute(process[pid].current);};     /* execute for about 4 months
            (void) wait(proc_repair);                  /* wait about 3 hrs for repair
            copy(process[pid].current,process[pid].initial,MANY); /* reset to initial state
            while (message_get(msg,status) {};         /* read and discard all msgs in queue
      };  };                                           /* bottom of work, break, fix loop
```


3.7.3.3 Reliable Processes via Checkpoint-Restart

So far, the processes described are failfast: they either execute the next step, or they fail and reset to the null state. They are not reliable because they do not repair their state upon failure. This failure behavior is similar to the storage failure model in which stores return to service in the null state. Some mechanism is needed to repair the process state, that is, to bring it up to its most recent value.

There are many ways to build such reliable processes from failfast processes. For simplicity, assume that every process' state fits within a single page. It is then easy to generalize to larger processes.

The following discussion proceeds by example, showing three different ways to implement the same server process. The server's job is to return a unique sequence number, called here a *ticket*, to each client upon request. It must never return the same ticket number to two different client requests. Clearly, to achieve this the server must reliably remember the most recently granted ticket number.

The simplest design is the *checkpoint-restart* approach. The process keeps two copies of its state in reliable storage. At each state change, it overwrites the oldest state value (ticket number) in reliable storage. At restart, it reads both state values and resumes execution from the most recent one (see Figure 3.19). The following code gives a simple example of a reliable checkpoint-restart process.

```
/* A checkpoint-restart process server generating unique sequence numbers for clients.   */
checkpoint_restart_process()                 /*                                         */
    { Ulong   disk = 0;                      /* a reliable storage group with state     */
      Ulong   address[2] = {0,1};            /* page address of two states on disk      */
      Ulong   old;                           /* index of the disk with the old state    */
      struct  { Ulong   ticketno;            /* process reads its state from disk.      */
                char    filler[VSIZE];       /* newest state has max ticket number      */
              } value [2];                   /* current state kept in value[0]          */
      struct  msg{                           /* buffer to hold input message            */
                processid   him;             /* contains requesting process id          */
                char        filler[VSIZE];   } msg;   /* reply (ticket num) sent to that address */
    /* Restart logic: recover ticket number from persistent storage                     */
      for (old = 0; old<=1, old++)           /* read the two states from disk           */
          { if (!reliable_read(disk,address[old],value[old] ))   /* if reliable read fails  */
              panic(); };                    /* then failfast                           */
      if (value[1].ticketno < value[0].ticketno) old = 1;   /* pick max seq no and determine */
          else { old = 0; copy(value[0], value[1],VSIZE);};  /* which value is old.     */
    /* Processing logic: generate next number, checkpoint, and reply                    */
      while (TRUE)                           /* do forever                              */
          { while (! get_msg(&msg)) {};      /* get next request for a ticket number    */
            value[0].ticketno = value[0].ticketno + 1;   /* increment ticket number     */
            if ( ! reliable_write(disk,address[old],value[0])) panic(); /* checkpoint   */
            old = (old + 1) % 2;             /* use other disk for state next time      */
            message_send(msg.him, value[0]);   /* send the ticket number to client      */
      };  };                                 /* endless loop to get messages.           */
```

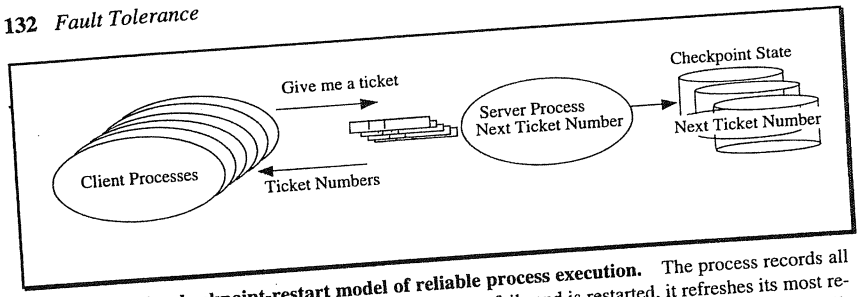

Figure 3.19: The checkpoint-restart model of reliable process execution. The process records all state transitions in the checkpoint state. When the process fails and is restarted, it refreshes its most recent state from the checkpoint-restart state. Such repair and restart yield reliable but not highly available processes, since repair times can last for hours.

Checkpoint-restart is a classic technique that has been widely used for decades (see Figure 3.1). Its major problem is that the repair time for the process can be quite long, since the process does not resume service until it is started and refreshes its state from persistent storage. The total repair time is therefore the process repair time plus the restart time, which, according to the model, is several hours altogether.

Because of the long repair times, checkpoint-restart yields highly reliable process execution, but not highly available process execution. The goal is to continue the process execution immediately after the process faults, meaning instant (almost zero) MTTR of process faults. If the fault is caused by hardware, it is likely that pair-and-spare hardware or triplexed hardware can mask it. If the fault is caused by software, by operations, or by environment (as most faults are), then the process and processors will fault, and the process will stop. Why not have a second process in a second processor take over from the first if the first process fails? That idea, called process pairs, is the next technique.

3.7.3.4 Reliable and Available Processes via Process Pairs

The *process pair* technique is somewhat analogous to *n*-plexed storage; it is a collection of *n* processes dedicated to providing a service. One of the processes is *primary* at any time. The primary process delivers the service. If the primary fails, the backup process continues the service. Clients send their requests to the primary, and if the primary fails, the new primary informs the clients of its new role.[9]

The two-process case, *process pairs,* is subtle enough. It allows a second process to *take over* from the primary. There are two delicate aspects of takeover:

Detection. Knowing that the primary is dead (being repaired).

Continuation. Knowing the most recent state of the primary.

As with all aspects of fault tolerance, detecting the primary failure is purely statistical. Because the primary can send messages only when it *is* operating, a failed primary cannot

[9] There is a generalization of process pairs to groups of *n* processes. But, as we will see, process pairs are close to or beyond the complexity barrier. In addition, the MTTF equations suggest that process pairs offer very long MTTF (≈30,000 years). For this reason the *n*-plex generalization is not considered here.

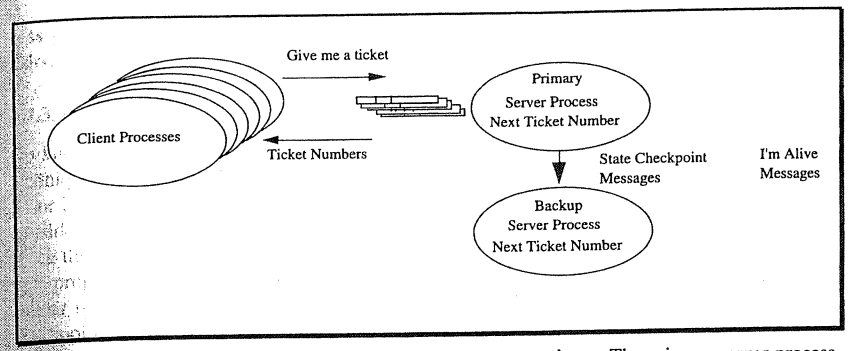

Figure 3.20: The process-pair model of reliable process execution. The primary server process sends all its state transitions to the backup process. When the primary process fails, the backup takes over, executing in the most recent state. This can yield takeover times measured in microseconds or milliseconds, providing high availability. The backup detects primary failure by noticing that no *I'm Alive* messages have arrived from the primary in a long time.

inform the backup that the primary is broken. For this reason, the primary sends *I'm Alive* messages to the backup on a regular basis. If one *I'm Alive* message fails to arrive on time, it is assumed to be delayed or damaged. But if several *I'm Alive* messages fail to arrive within a reasonable time, the backup assumes that the primary did not send them and is being repaired.

Suppose the backup has decided that the primary is dead. What should it do? How can it continue the computation from the most recent primary state? First, it needs to know the most recent primary state. The usual technique is for the primary to combine state-change messages to the backup with the *I'm Alive* messages, so-called *checkpoint messages.* For example, in Figure 3.20 the *I'm Alive* messages are being sent frequently to the backup. Clearly, the state-change messages can piggyback on the *I'm Alive* messages, or they can act as surrogates for them. The basic logic of a process-pair program is shown in Figure 3.21.

The idea presented in Figure 3.21 is that both processes start out in the restart state. One process (say, the even-numbered one) is the default primary and takes priority by waiting only one second to see if the other is alive. After two seconds, either the default primary process will have generated an *I'm Alive* message to the default backup, or the backup decides the primary is dead. In that case, the backup becomes the primary. In any case, whichever becomes the primary broadcasts its identity to all concerned and then goes into a loop of looking for input and processing it. This loop continuously sends *I'm Alive* messages to the backup, whether there is any input or not. For its part, the backup sleeps for a second, looks to see what the new state is, and then goes back to sleep. The "newer state" test is designed to avoid the possible permutation of messages in the input queue by ignoring states older than the current backup's state. This test also discards duplicate messages. If the backup does not get a new state within a second, then the primary must be dead, since the primary ought to be able to send hundreds of *I'm Alive* messages in that time. If the primary is dead, the backup broadcasts that it is the new primary and waits for requests.

There is one more delicate takeover issue. At takeover, the backup is in the same state the primary was in when it sent the checkpoint message to the backup. But the primary may have failed before sending the reply that the last state transition generated (the bottom box of

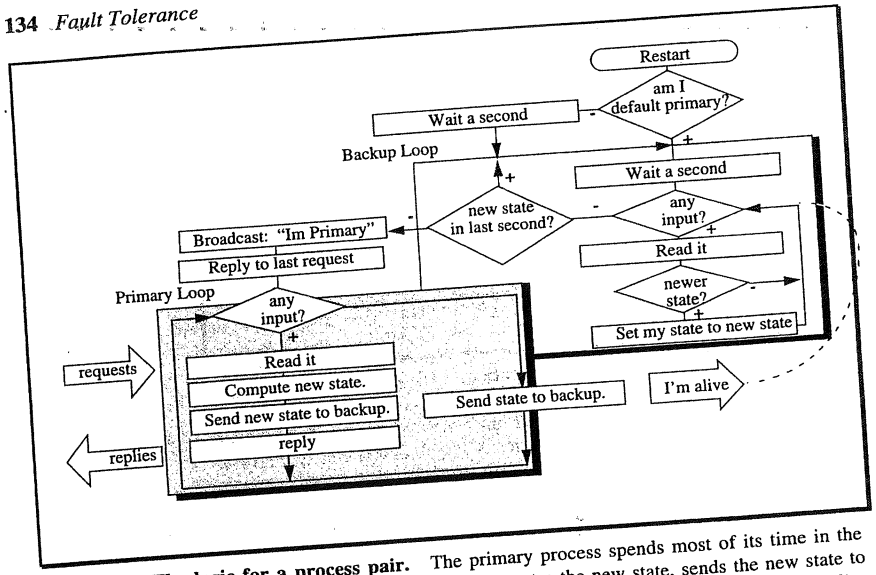

Figure 3.21: The logic for a process pair. The primary process spends most of its time in the shaded box (primary state), where it reads requests, computes the new state, sends the new state to backup, and then replies. If it has nothing else to do, the primary simply sends out an *I'm Alive* message, presumably generating thousands per second. The backup, executing exactly the same program, consumes these *I'm Alive* and state checkpoint messages. If no messages from the primary arrive at the backup for one second, the backup assumes the primary is dead and takes over the responsibilities of the primary.

Figure 3.21). So, if a reply was owed to a client, the reply is resent. This may generate a duplicate message to the client, but that was a possibility of the message system anyway.

To continue the discussion of Figure 3.21: If the maximum propagation time for messages is one millisecond, and the maximum service time for a request is a millisecond, then waiting ten milliseconds (five message cycles) is long enough to detect process failure. This period is long enough for five consecutive lost *I'm Alive* messages if processors are dedicated to processes. Thus, process takeover times of ten milliseconds can easily be achieved. Typically, though, the primary shares the processor with other processes and, for resource consumption reasons, does not continuously send *I'm Alive* messages. Consequently, a longer latency is more typical.

In fault-tolerant operating systems, process pairs are a basic notion provided by the system. Such operating systems inform the backup process when the primary process fails and automatically redirect all new client requests to the backup process. This means that the primary can dispense with the idle state checkpoint messages and the *I'm Alive* messages. It also means that the backup can dispense with the one-second waits, because all the messages are important. The backup can also dispense with the broadcast to clients at startup and takeover. These are just performance optimizations; the programming style of checkpointing state changes is the same for this simple model and for the "real" fault-tolerant systems.

Adapting the checkpoint_restart_process() example to process_pair_process() using the logic of Figure 3.21 is straightforward (about one page of C code). The complexity of process pairs apparent in the above explains why the discussion here is restricted to process pairs, rather than including *n*-plexed processes.

3.7.3.5 The Finer Points of Process-Pair Takeover

No known technique continues process execution immediately after a fault.[10] The known techniques offer takeover times on the order of a few message propagation delays. In a cluster or local network, this takeover can happen in milliseconds when software overheads are considered; in long-haul nets, the message propagation delays imply that takeover times will be on the order of tens or hundreds of milliseconds (see Section 2.9 for a discussion of message propagation times). To clients operating on a time scale of seconds, this takeover may appear immediate. Certainly, it is a big improvement over the availability offered by the checkpoint-restart approach, which has outages measured in minutes or hours.

There are a few additional details. If state-change checkpoint messages were permuted in the backup process' input queue, then an old state might replace a newer one. The permuted messages problem is easily solved by having the primary process sequence number the state-change messages, and by having the backup ignore old ones—ones with sequence numbers less than the current sequence number. The next problem is much more serious. If the checkpoint messages are lost *and* the primary fails, then the backup will not be in the correct state. At this point, one must appeal to the single-fault tolerance assumption. These are process pairs; they tolerate single faults. If both a process and the message fail, then there is a double failure, which is not protected. Assuming independence of failures, the mean time to such failures is approximately MTTPF/pmf, the time to a process failure over the chance that the most recent message was lost. For the numbers chosen in the process and message model above, that is ($\approx (10^7/10^{-5}) = 10^{12}$) seconds, or about thirty-thousand years. If that answer is not acceptable, then a safer approach is to have the backup acknowledge receipt of the state change before the primary responds to the client. But now the primary has to sense backup failures (it has to know when to stop waiting for the backup to acknowledge). This acknowledged-checkpoint strategy is the one used in the Tandem Computer's process pairs; for the sake of simplicity, however, it is not the one described here.

The ticket-number generator program was used to demonstrate checkpoint-restart logic. This three-line program is not very demanding, either as a checkpoint-restart process or as a process pair. Its only requirement is that it never generate the same ticket number twice. As such, it fails to expose one of the most delicate issues in takeover. Suppose the primary is performing some real operation, such as advancing the rods in a nuclear reactor by one inch, and then it fails. The backup may be in a quandary: did the primary perform the operation or not? The backup cannot tell unless it "looks" at the rods. Sometimes, though, it is not possible to "look."

A whole vocabulary has been built up around this problem. First, an operation is *testable* if the program can reliably establish whether the operation has been performed and

[10] One scheme, called atomic broadcast, sometimes works. In atomic broadcast, the client sends requests to N different servers and accepts the first response. This works well if all requests to the servers commute (if ordering is not important). Read-only servers have this property. In general, however, requests do not commute. Thus, if two clients broadcast their requests to *n* servers, the servers may process the requests in different order. This can produce different final states and different answers. For this reason the *n*-server design forces the servers to agree to an ordering of the requests in the general case. Such an ordering mechanism is called *atomic broadcast*. It synchronizes the servers and so introduces delays in transaction (request) processing.

the desired effect achieved. Writing pages to disk is testable (they can be read to see if they have changed). Moving reactor arms to a particular position is testable, but movement to a relative position is not: "move to position X is testable," while "move 1 inch" is not testable. Writing to a display or printing on paper is not testable. In general, many physical operations are not testable. The safe thing, then, is to redo the operation: move the reactor arm to position X again, write to the display again, print the message again, and write to the disk again. But not all operations can safely be redone. An operation is *restartable* or *idempotent* if redoing it is safe; that is, if redoing it an arbitrary number of times is equivalent to doing it once. Moving the reactor rods one inch is not restartable, printing a check is not restartable (n operations are likely to print n checks), and inserting a record at the end of a file is not restartable (it may insert the record n times); but moving the rods to position X, writing a disk page, and writing a display are probably idempotent operations.

In Figure 3.21, resending the reply message at takeover is trying to deal with the idempotence issue. The takeover process cannot tell whether the client received the reply to his request, so the server resends the reply on the premise that the message will be discarded by the client if it is a duplicate. The operation is not testable, but it is assumed to be idempotent.

3.7.3.6 Persistent Processes—The Taming of Process Pairs

As can be seen from the previous discussion, writing process pairs is very demanding. Indeed, it is the authors' experience that everyone who has written one thinks it is the most complex and subtle program they have ever written. However, reliable systems use these techniques, and those systems do mask many failures.

There is a particularly simple form of process pair called a *persistent process pair.* Persistent process pairs have a property that is variously called *context free, stateless,* or *connectionless*. Persistent processes are almost always in their initial state. They perform server functions, then reply to the client and return to their initial state. The primary of the persistent process pair does not checkpoint or send *I'm Alive* messages, but just acts like an ordinary server process. If the primary process fails in any way, the backup takes over in the initial state.

Persistent process pair takeover works as follows. The underlying fault-tolerant operating system knows that the primary and backup are a process pair. If the primary fails, or if the processor on which the primary is executing fails, the operating system of the backup process senses the failure and informs the backup process of the failure. The operating system also redirects all client messages to the backup. In this situation, the backup takes over in its initial state and begins offering service.

As described, persistent processes are easy to program and make high-availability servers, but these servers seem to provide little more than encapsulation of the server logic to the client. The servers do not maintain network state or database state. To make persistent processes useful, someone else must maintain such states.

For example, imagine that such a persistent process acted as your mailbox. It would receive messages for you, but would always return to the null state (no messages for you)—not a very useful program. Similarly, if the server actually kept your state in a file, the updates to receive or discard a mail message might be partially incomplete at the time the server failed. In this case, your mailbox file would be left in that state by the backup persistent process when it took over in the null state.

These examples hint at the solution. A persistent process server should maintain its state in some form of transactional storage: a transaction-protected database. When the primary process fails, the transaction mechanism should abort the primary's transaction, and the backup should start with a consistent state. On the other hand, if the server calls commit_work() after making the changes, the transaction mechanism should reliably deliver the server's message to the client as part of the atomicity guarantee of ACID transactions.

Given that, the transaction mechanism returns the database and the network to their state as of the start of the server function. Persistent processes need not send application checkpoint messages to indicate state changes, since the initial state is the takeover state. When running on a fault-tolerant system that provides persistent process pairs as a primitive, the program need not concern itself with takeover or message retransmission. That logic is handled for the program by the underlying system. All this makes persistent process pairs very easy to program. They have the following form:

```
persistent_process()                    /* prototypical persistent process                    */
  { wait_to_be_primary();               /* wait to be told you are primary                    */
  while (TRUE)                          /* when primary, do forever                           */
    { begin_work();                     /* start transaction or subtransaction                */
    read request();                     /* read a request                                     */
    doit();                             /* perform the desired function                       */
    commit_work();                      /* finish transaction or subtransaction               */
    reply();                            /* reply                                              */
    };                                  /* did a step, now get next request                   */
  };                                    /*                                                    */
```

The following code illustrates the checkpoint-restart ticket server of Subsection 3.7.3.3 reprogrammed as a transactional server.

```
/* A transactional persistent process that acts as a server generating unique tickets for clients */
persistent_ticket_server()                          /* current state kept in sql database          */
  { int    ticketno;                                /* next ticket number (local variable from DB) */
  struct   msg{                                     /* buffer to hold input message                */
           processid    him;                        /* contains requesting process id              */
           char         filler[VSIZE];              /* reply (ticket number) sent to that addr     */
           } msg;                                   /*                                             */
  /* Restart logic: recover ticket number from persistent storage                                   */
  wait_to_be_primary();                             /* wait to be told you are primary             */
  /* Processing logic: generate next number, checkpoint, and reply                                  */
  while (TRUE)                                      /* do forever                                  */
    { begin_work();                                 /* begin a transaction                         */
    while (! get_msg(&msg));                        /* get next request for a ticket number        */
    exec sql update ticket set ticketno = ticketno + 1; /* increment the next ticket number        */
    exec sql select max(ticketno)                   /* fetch current ticket number                 */
             into :ticketno                         /*   into program local variable               */
             from ticket;                           /*   from SQL database                         */
    commit_work();                                  /* commit transaction                          */
    message_send(msg.him, value);                   /* send the ticket number to client            */
  }; };                                             /* endless loop to get messages.               */
```

Notice that to become a process pair, the ticket server application did nothing more than represent its state in transactional storage and call the operating system routine

wait_to_be_primary(). There are no checkpoint messages, no *I'm Alive* messages, and no takeover logic. Yet this process has much higher availability than the checkpoint-restart process. In addition, if the transaction did complex updates to its database, each group of updates (transactions) would have all the ACID properties.

3.7.3.7 Reliable Processes plus Transactions Make Available Processes

Transactions provide highly reliable execution, but ACID says nothing about availability. All the ACID properties are reliability properties. Process pairs provide highly available execution (processes), but they are hard to program because takeover is so delicate. The transaction ACID properties provide clean takeover states: transactions combined with persistent process pairs give both high reliability and high availability by allowing simple programs to use process pairs.

Virtually all fault-tolerant application-level process pairs are now programmed as persistent processes. Only low-level processes (device drivers, disk servers, TP monitors, and the transaction mechanism itself) are programmed as raw process pairs. These more basic processes cannot use the persistent process approach, primarily because they are below the level of abstraction that provides the high-level process-pair and transaction mechanisms.

Highly available systems contain many process pairs. The storage repair processes mentioned previously need to be process pairs. If the storage is shared and access to it must be serialized, then the storage servers should be process pairs. If reliable storage is to be implemented, then the reliable writes should indeed be reliable (write *all n* of the *n*-plex). To protect against faults within the reliable write code, it should be run as a process pair, and so on.

Process pairs depend critically on the premise of single failure (only one processor fails during the repair period. If the pair of processes has a shared power supply (or shared memory or any other single point of failure that can disable them both), then this assumption is violated. The processes must therefore be connected to each other, to their storage, and to their clients via dual networks. They must have independent power supplies and, of course, they must have different processors and memories. In addition, each component should have independent failure modes and be failfast. It is fair to say that today most hardware systems are not constructed to meet these demands. Several attempts to build process-pair systems have failed. Yet there are two successful examples, Tandem's Guardian system and IBM's XRF system. The XRF design demonstrates that it is possible to retrofit these ideas to a pre-existing general-purpose transaction processing system (IMS in this case).

In summary, given failfast processes that fail independently, there are two ways to get highly reliable processes (checkpoint to storage or checkpoint to a backup process) and one way to get available processes (process pairs). The model allows us to talk about and analyze both.

3.7.4 Reliable Messages via Sessions and Process Pairs

Having shown how to convert faulty processes and faulty storage to highly available processes and storage, the discussion turns to converting faulty messages to highly reliable messages. This section first defines the behavior and fault model for messages. It then shows how sessions, session sequence numbers, acknowledgment, timeout, and retransmission can all be combined to provide reliable message delivery. Since either the message source process or the destination process can fail, process pairs are part of the mechanism needed to convert faulty messages into reliable messages.

3.7.4.1 Basic Message Model

A process can send a message to another process by invoking message_send(), which specifies the process identifier and the message data. This copies the message data from the sender's address space to the receiver's input queue. The receiver can then call message_get() to copy the message from its input queue to its address space.

The message and process queue definitions were already given. Here is the code to implement message_send and message_get:

```
/*send a message to a process:      returns true if the process exists                    */
Boolean message_send(processid him, avalue value)  /*                                    */
        { amessagep      it;                       /* pointer to message created by this call */
        amessagep  queue;                          /* pointer to process message queue   */
        if (him > MAXPROCESS) return FALSE;        /* test for valid process             */
loop:   it = malloc(sizeof(amessage));             /* allocate space to hold message and fill */
        it->status = TRUE; it->next = NULL;        /* in the fields                      */
        copy(it->value,value,VSIZE);               /* copy message data to message body  */
        queue = process[him].messages;             /* look at process message queue      */
        if (queue == NULL) process[him].messages = it; /* if the queue is empty then     */
        else                                       /* place this message at head of queue */
          {while (queue->next != NULL) queue = queue->next; /* else find end of queue   */
          queue->next = it;}                       /* and place the message at queue end . */
        if (randf() < pmf) it->status = FALSE;     /* sometimes message is corrupted.    */
        if (randf() < pmd) goto loop;              /* sometimes the message is duplicated */
        return TRUE;                               /*                                    */
        };                                         /*                                    */

/* get the next input message of this process:      returns true if there is a message */
Boolean message_get(avalue * valuep, Boolean * msg_status)/*                           */
        { processid    me = MyPID();               /* caller's process number            */
        amessagep   it;                            /* pointer to input message           */
        it = process[me].messages;                 /* find caller's process message queue */
        if (it == NULL) return FALSE;              /* return false if queue is empty     */
        process[me].messages = it->next;           /* take first message off the queue   */
        *msg_status = it->status;                  /* record its status                  */
        copy(valuep,it->value,VSIZE);              /* value = it->value                  */
        free(it);                                  /* deallocate its space               */
        return TRUE;                               /* return status to caller            */
        };                                         /*                                    */
```

Messages have simple failure modes; some fraction of them spontaneously decay (status = FALSE) and some fraction of them are duplicated. This models failure of communication

lines, buffer overflows, and the like. It also models the retransmission logic underlying message systems that occasionally duplicate a message. In addition, the messages in a process's input queue may be arbitrarily permuted. This models the possibility that messages can be arbitrarily delayed in the network.

3.7.4.2 Sessions Make Messages Failfast

As defined, messages are not failfast; corrupt and duplicate messages are delivered, and messages are delivered out of order. The first step in making messages reliable is to give them simple failure semantics—failfast semantics. To make all messages failfast, faulty messages are converted to lost messages. Message semantics, then, are as follows: (1) messages are delivered in order, and (2) some messages are lost. In the simplified model, the only form of message failure is a lost message.

First, let's convert corrupt messages to lost messages. The message status flag models the idea that the message contents are self-checking, so that corrupt messages can be detected and discarded. It is standard to compute a message checksum at the source and to send the checksum as part of the message. The receiver recomputes the checksum and compares it to the sender's checksum. If the two checksums are different, the message has been corrupted and is discarded. This converts a corrupt message to a lost message. The get_msg() procedure defined in Subsection 3.7.3.1 discards all such corrupt messages.

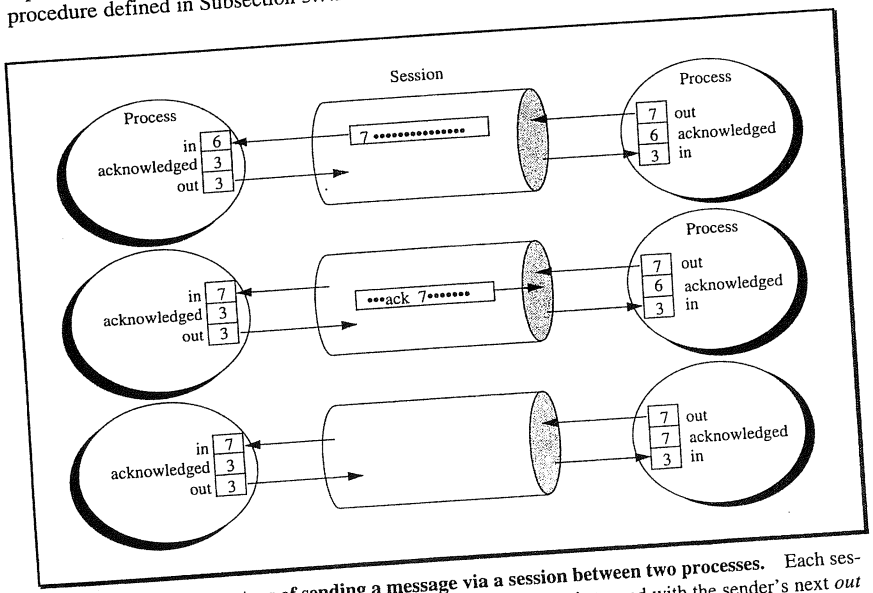

Figure 3.22: The three steps of sending a message via a session between two processes. Each session endpoint maintains three sequence numbers. A new message is tagged with the sender's next *out* sequence number. The receiver increments its *in* sequence number when the next sequential message is received. The receiver sends an acknowledgment message for that sequence number. When the sender gets the acknowledgment, it increments its *acknowledged* sequence number. This mechanism detects duplicate and delayed messages.

The simplest approach to detecting and discarding duplicate and permuted messages is to use sessions and message-sequence numbers. A *session* is a bidirectional message pipe between two processes. Sessions have the following semantics: (1) messages are delivered to the destination in the order they were sent, and (2) no messages are duplicated or lost, but messages can be delayed. Sessions are variously called connections, sockets, pipes, dialogs, opens, ports, and myriad other names. In this text, sessions are represented by pipe icons, as shown in Figure 3.22.

Sessions implement these semantics by having the two session endpoints each maintain three sequence numbers: (1) the *out* sequence number, which is the sequence number of the most recently sent message on that session, (2) the *in* sequence number, which is the sequence number of the most recently received message on that session, and (3) the *acknowledged* sequence number, which is the sequence number of the most recently acknowledged message.

Each message from one process to another carries a sequence number. When a new message is sent to that receiver, the sender's *out* sequence number is incremented and used to tag the sender's message. When the receiver gets the message, if the sequence number is one greater than the most recent message, the recipient knows it is the next message and accepts it. The recipient increments the *in* sequence number and sends an (unsequenced) *acknowledgment message* to the sender, saying that the new message arrived.

Sequence numbers detect duplicate and delayed messages as follows: If a message is duplicated, the message sequence number is also duplicated; thus the recipient can detect a duplicate message by noticing that the message sequence number is equal to the most recently acknowledged sequence number. In this case, the recipient discards the message and does not change its *in* sequence number. In either case, the recipient acknowledges the message to the sender.

If a message is delayed and hence arrives out of order, the recipient can see that the message sequence number is less than the most recently acknowledged message and consequently can discard the message. It need not acknowledge such messages.

As described so far, sessions create the simple failure semantics of sending messages in order, thereby making messages failfast. If there were no failures, such sessions would work perfectly. But, as soon as there is a message failure, the session starts discarding all future messages in that direction. So far, then, the design has made messages failfast. The next step is to make messages reliable by repairing lost messages.

3.7.4.3 Sessions plus Retransmission Make Messages Reliable

To restate the session message-sending protocol: When one process sends a message to another, the sender increments the session *out* sequence number and tags the message with the session name (the names of the two processes) and the sender's current *out* session sequence number. The sending process then waits for an acknowledgment message, called an *ack*, from the recipient. The ack message carries the session name and the original message sequence number. Because message failures cause messages to be lost, the acknowledgment may never arrive. In such cases, the sender must resend the message.

To repair lost messages, the sender has a *timeout* period. If the sender does not receive an ack within a timeout period, it *retransmits* the message and continues to do so until an ack is received. If the sender and receiver are functioning, the message eventually is delivered, and the ack is delivered. When that happens, the sender declares the message delivered and increments its acknowledged sequence number.

3.7.4.4 Sessions plus Process Pairs Make Messages Available

The discussion so far has focused on message failures and ignored process failures. What if the sending or receiving process fails? In that case, the message has nowhere to go or no sender to retransmit it. A session is only as reliable as its two endpoints.

The obvious solution to making reliable session endpoints is to make each endpoint a process pair. In this design, sessions connect process pairs so that delivery of new messages to the backup is automatic if the primary fails. Messages that have already been sent to the primary are probably lost and will not arrive at the backup. The sequence numbers of the primary are also maintained as part of the backup process state. If a process fails, that endpoint of the session switches to the backup of that process pair. The backup will have the current sequence numbers, and if it is sending, it will have a copy of the current message.

The logic for this is simple: The sender first checkpoints the message to the backup and then sends the message. If the backup takes over, it resends the message and continues the session. The receiver checkpoints received messages to its backup before acknowledging them. This logic is diagramed in Figure 3.23.

At process-pair takeover, the backup process broadcasts its new identity and resends any unacknowledged messages that the primary process was trying to send. The other session endpoint may get a takeover broadcast from the backup process of the process pair. In that case, the other endpoint begins sending the message to the former backup process,

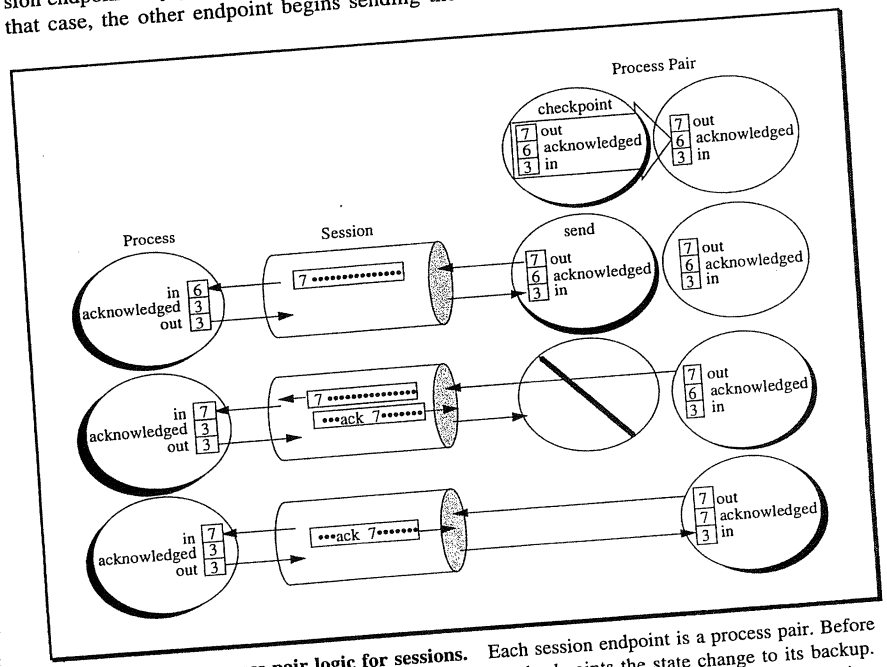

Figure 3.23: The process pair logic for sessions. Each session endpoint is a process pair. Before sending a message or an acknowledgment, the process checkpoints the state change to its backup. When a process fails, the session endpoint switches to the backup process that takes over the session. Now all messages flow to the backup.

which is now the new primary process of the process pair. When the message is finally delivered and acknowledged, the next message can be sent via the session.

To restate this development: Session sequence numbers convert permuted messages and duplicate messages into lost messages, making all messages failfast and making sessions failfast-communications services. Acknowledgment, timeout, and retransmission make messages reliable by resending them. Messages eventually get through. Process pairs make the endpoints available; they are persistent in sending the messages until they are acknowledged. Messages are as available as the processes that send them and the wires that carry them. If the processes are duplexed, and the wires are *n*-plexed, the messages can be made highly available. The various time constants (acknowledge times and takeover times) are limited by message delay and processor speed. It is possible to give reliable delivery within seconds for wide-area systems (≈10,000 km) and within tens of milliseconds for local area systems (1 km). These time constants are discussed in Section 2.6.

3.7.4.5 The Code for Sessions and Messages

The code to implement sessions, acknowledgment and retransmission, and process-pair checkpoint and takeover is instructive. First, sessions and message formats are defined, and an array of sessions is added to the state of each process. These sessions hold the endpoint names and the sequence numbers.

```
/* definition of a session address, the source and destination processes (or process pairs)   */
typedef struct    Session * asessionp;          /* typedef for session pointer                */
typedef struct    Session{                      /* typedef for sessions                       */
                  processid    source;          /* source process (or pair) of session (sender) */
                  processid    target;          /* destination process of session (receiver)  */
                  Ulong        in;              /* sequence number of most recently acked msg */
                  Ulong        out;             /* sequence number of most recently sent msg  */
                  Ulong        ack;  } asession; /* sequence number of most recent ack        */
enum MSG_TYPE { MSG_NEW,MSG_ACK,MSG_TAKEOVER}; /*                                            */
typedef struct Session_message * a_session_messagep; /* typedef for session pointer           */
typedef struct Session_message{                 /* typedef for messages flowing on sessions   */
                  asession     session;         /* a msg has session name plus sequence nums  */
                  MSG_TYPE     type;            /* message type: data, ack, takeover.         */
                  avalue       value;           /* data if this is a data message             */
                  } a_session_message;          /*                                            */
/* each process has an array of sessions and for each session a list of input messages        */
asession     my_session[MAXPROCESS];            /* sessions belonging to this process         */
amessagep  in_msg_queue[MAXPROCESS];            /* process queue of acked input messages      */
void Initialize_sessions(void)                  /*                                            */
   { int him;                                   /*                                            */
   for (him = 0; him<MAXPROCESS; him++)         /* initialize global structures               */
      {my_session[him].source = MyPID();        /* session is from my process id              */
      my_session[him].target = him;             /* to his process id                          */
      my_session[him].in = my_session[him].ack = my_session[him].out = 0;  /* seq numbers start at zero */
      in_msg_queue[him] = NULL;                 /* process input queue is null initially      */
   };  };                                       /* end of initialization of process session arrays */
```

In addition, assume that the message_send() and message_get() routines have been abstracted to session_send() and session_get() routines. These routines send and get messages that carry a session descriptor and a message type (that is, they manipulate the a_session_message type rather than just a_message type). These routines pack and unpack the session descriptor and message type descriptor as added framing information on messages. Session_send() fills in the session descriptor and value for each message. Session_get() extracts these two fields from the next process input message and returns them. Session_get() must read the message to find the session name and the sequence number. If the message is corrupt (status = FALSE), session_get() cannot read these fields and discards such messages. Consequently, it has no status parameter; any message it returns is not corrupt. These are the interfaces:

Boolean **session_get(asession *session, MSG_TYPE *type, avalue * value);**
Boolean **session_send(asession session, MSG_TYPE type, avalue value);**

The reliable_send_message() routine sends the message, waits for the timeout period, and if no acknowledgment message has arrived, it sends the message again. It continues this until the message has been acknowledged (the acknowledgment mechanism is explained momentarily):

```
const Ulong  timeout = 1;                                  /* sender's timeout for ack wait
void reliable_send_message(processid pid, avalue value)    /* send a message reliably
    { asessionp sessionp= &my_session[pid];                /* point at the session to that process
    sessionp->out = sessionp->out + 1;                     /* increment session out sequence number
    checkpoint(*sessionp,value);                           /* checkpoint session and msg to backup
    do                                                     /* keep sending until msg is acknowledged
        {session_send(*sessionp, MSG_NEW, value);          /* send
        wait(timeout);}                                    /* wait a second
    while (sessionp->out != sessionp->ack);                /* repeat until message is acknowledged
    };                                                     /*
```

In this design, the sender continues sending until the receiver acknowledges the message. The receiver may be doing something else at the moment—computing, writing storage, or even sending a message to some third process. The receiver, therefore, may not be interested in acknowledging the sender's message right away. Waiting for the receiver's acknowledgment might cause message deadlock, with each process waiting for an acknowledgment from the next. To avoid such deadlocks, and to speed message flows, acknowledgments should be generated quickly and asynchronously from the receiver's application program logic. This means that each process should regularly poll its input sessions for messages. Such a requirement, however, would make programs both unreadable and error-prone.

Even more fundamental, perhaps, is the fact that the sender in the code above assumes someone else is going to update the acknowledgment sequence number. The reliable_send_message() code could perform this logic, but it would vastly complicate the code. In general, this is the kind of logic that another process should manage as a service for all processes in the address space.

To preserve the program structure, almost all systems use the concept of multiple processes sharing one address space to perform such asynchronous tasks (see Chapter 2). Such lightweight processes are often called *threads,* but the generic term *process* is used here. These processes execute continuously and autonomously from the main application, communicating with it primarily via shared memory. The main application function is implemented as one or more processes (threads), and service functions such as message acknowledgment are implemented as other processes.

Each address space has a *listener* process that performs message acknowledgment and increments message sequence numbers for the application process executing in that address space. The listener is constantly polling the application process's sessions. When a message arrives, the listener executes the sequence number logic. If the message is a data input message, the listener acknowledges the message and places it in the session's message queue for the application program process. If the message is an acknowledgment, the listener implements the ack sequence number logic. Figure 3.24 illustrates the role of the listener .

The listener process executes continuously, running at high priority. When there are no input messages, it waits for the next event (in the sample code it waits for a second rather than for an event). In most designs, the listener places a limit on how many messages it will accept from a sender. This is called *flow-control.* Otherwise, senders could flood the receiver with messages, consuming all the listener's storage. For simplicity, such refinements are not included in the example. Also for simplicity, the code does not deal with takeover (process pairs). The listener maintains the process sequence numbers and the FIFO session message queues.

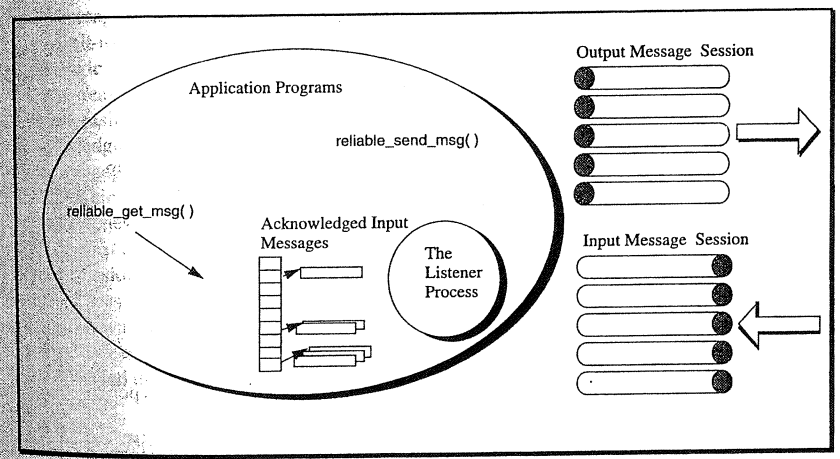

Figure 3.24: The listener process manages message acknowledgment for the processes executing in the address space. It executes asynchronously within the address space of the application process. Its job is to quickly acknowledge input messages arriving via the input sessions, adding them to the process input queue. It also manages the session sequence numbers and checkpoints changes to the backup process of the process pair.

```
/* The listener process runs in the background of each address space. It acknowledges input
/* messages and records the acknowledgement of sent messages                              */
void listen(void)
    {asession       session;                    /*                                       */
    processid       him;                        /* the session the message arrived on    */
    MSG_TYPE        type;                       /* identity of message sender            */
    Ulong           seqno;                      /* type of message (data, ack, other)    */
    avalue          value;                      /* sequence number of message            */
    amessagep       it;                         /* contents of message                   */
    amessagep       queue;                      /* pointer to new msg added to queue     */
    initialize_sessions();                      /* pointer to session message queue      */
    while (TRUE)                                /* initialize global structures          */
        { while (session_get(&session,&type,&value))  /* do forever: read msgs and ack them */
            { him = session.source;             /* get message, if none wait a second    */
            seqno = session.out;                /* who sent the message                  */
            switch (type)                       /* extract message sequence number       */
                {case MSG_NEW:                  /* what kind of message                  */
                    { if (my_session[him].in == (seqno - 1))  /* a new data message      */
                        {it = malloc(sizeof(amessage))        /* if it is the next message */
                        copy(it->value,value,VSIZE);          /* get and format the message block */
                        it->next =NULL;                       /* copy message to queue element */
                        if (in_msg_queue[him] == NULL)        /* it is at the end of the enqueue */
                            in_msg_queue[him] = it;           /* if queue is empty add  */
                        else { for(queue = in_msg_queue[him] ; queue->next != NULL ; ); /*
                                queue = queue->next;          /* element to queue head  */
                            queue->next = it;};               /* else go to end of the queue */
                        my_session[him].in ++;                /* and insert new message */
                        checkpoint(my_session[him],value);} /* ckpt session and message to backup */
                    session_send(session,MSG_ACK,value);} /* increment sequence number */
                case MSG_ACK:                   /* ack message in any case               */
                    {if (session.out == my_session[him].out) /* got an ack for a message */
                    my_session[him].ack ++;}    /* if it ack's most recent send          */
                case MSG_TAKEOVER:              /* increment acknowledged seqno          */
                    {};                         /* process pair takeover message         */
                default: {};                    /* logic not handled in this example     */
                };                              /* ignore message by default             */
            };                                  /* end of case switch on message type    */
        wait(1);                                /* end of get message loop               */
    };  };                                      /* nothing in input queue, wait a second */
                                                /* end of listener infinite loop.        */
```

The picture thus far is that the listener acknowledges receipt of messages promptly and saves these messages as part of the process (pair) state. The acknowledged input messages of each session are chained from the in_msg_queue[] of that session in most-recent-last order. It should now be clear that reliable_send_message() does reliably send messages. It retransmits every second until the message is acknowledged and the listener records that acknowledgment by incrementing the session.ack sequence number.[11]

Reliable_get_message() gets the next message value from a particular session by accessing the message queue maintained by the listener. The code to reliably get a message follows:

```
Boolean reliable_get_message(processid him, avalue value)   /* get a message                */
    { amessagep msg = in_msg_queue[him];                    /* points a first msg in queue  */
    if (msg == NULL) return FALSE;                          /* return false if queue empty  */
    in_msg_queue[him] = msg->next;                          /* remove msg from queue        */
    copy(value,msg->value,VSIZE);                           /* copy msg value (value = msg->value) */
    free(msg);                                              /* deallocate message storage block */
    return TRUE;                                            /*                              */
    };                                                      /*                              */
```

3.7.4.6 Summary of Messages

Reliable_get_message() and reliable_send_message(), combined with the listener process, convert ordinary messages into failfast messages. They discard corrupt messages, duplicates, and permutations. Sender timeout and retransmission until acknowledgment transform failfast messages into reliable messages. Process pairs make the session endpoints available, thereby transforming reliable messages into highly available messages. This sequence of steps shows how ordinary unreliable messages can be transformed into a reliable and available message service.

Fault-tolerant operating systems provide the reliable message get and send logic, the listener logic, and the logic to redirect messages to the backup of a process pair at takeover. For this reason, most application programmers are unaware of the acknowledgment-timeout-retransmission issues; they are even unaware of the checkpointing of the message sends and acknowledgments to the backup. But it is useful to have this simple model of how the underlying message system works.

One nice way to talk about messages is to say that raw messages may be delivered zero or more times. By adding sequence numbers and discarding duplicates, messages are delivered *at most once*. Such messages, however, may be delivered once or not at all. By adding retransmission, messages are delivered to a process *exactly once*. But this just means the message was delivered to the listener. Perhaps the message was never delivered to the application, or was never acted upon by the receiver. By adding process pairs, the message is *processed exactly once*. Consequently, these techniques give a simple form of message atomicity and durability (the A and D of ACID).

3.7.5 Summary of the Process-Message-Storage Model

The preceding gives a simple simulation model of the three components of a computer system. The message and storage models show how software can mask hardware faults by

[11] Both reliable_get_message() and reliable_send_message() share the session and message queue data structure with the listener process. Updates to this shared data structure must be synchronized by a semaphore or by the use of atomic machine instructions as described in Chapter 12. That issue is ignored here.

using redundancy in hardware, duplexing storage, wires, and processors. It also shows how software can use redundancy in time by retransmitting a message if the first one fails.

This material gives a sense of three styles of software fault tolerance: checkpoint-restart, process pairs, and persistent processes. It also gives a sense of the relative difficulty and benefits of the three styles, with the goal of convincing the reader that, if possible, persistent processes plus transactions should be used to obtain highly available program execution.

In addition, this simplified model (leave aside the actual implementation) is fairly complex. No formal or theoretical development of these ideas is available—it would be too difficult and complex to handle. In contrast, the model might work well for simulation studies to help quantify the relative benefits of design alternatives.

3.8 General Principles

We have all learned the following two rules, and done so by bitter experience:

KISS. Keep It Simple, Stupid!

Murphy. Whatever can go wrong will go wrong at the worst possible time and in the worst possible way.

Modularity, failfast, *n*-plexing, and ACID are all attempts to deal with these rules. Imagine programming a module that either works or misbehaves arbitrarily, and trying to tolerate the arbitrary misbehavior. That is the behavior predicted by Murphy's law. Failfast, also called all-or-nothing (atomicity), simplifies failure modes. Simplicity is what KISS is about. Don't be fooled by the many books on complexity or by the many complex and arcane algorithms you find in this book and elsewhere. Although there are no textbooks on simplicity, simple systems work and complex ones don't. Modularity, failfast, *n*-plexing, and ACID are mechanisms that simplify design, and make it easier to build systems with high functionality yet simple structure.

Given that things fail, and given that they fail fast, then repair is important to both reliability and availability. Process pairs, disk pairs, system pairs, and quick message retransmission all yield repair times measured in milliseconds—in effect, instant repair for people and physical processes that operate on time scales of seconds.

If faults are independent, then single-fault tolerance, combined with repair, yields MTTFs measured in decades or millennia. At these rates, other factors become significant (labor disputes, fires, floods, business panics, or whatever). The next improvement is several orders of magnitude away and is justified only rarely, generally only when human life is at stake.

Designing a fault-tolerant system needs a model of fault modes and their frequencies. The hardware and software design should minimize module interdependencies to get good fault containment. If faults are to be independent, the modules should have independent failure modes—suspenders and a belt, rather than two pairs of suspenders or two belts. This is sometimes called *design diversity*.

A model of fault tolerance needs to be *end-to-end.* It needs to look at the whole system, including operations and environment, to make estimates of system reliability and availability and to tolerate faults at the highest level of abstraction. The next section has an extended example of end-to-end problems.

The models used in the previous three sections are not end-to-end: They focus on the narrow topics of hardware or software fault tolerance. But they do allow careful discussion of the key fault-tolerance architectural issues. The message model and storage model of the previous section are relatively easy to understand and use. In contrast, the process pair model is intellectually challenging for the brightest people. As such, process pairs are a dangerous tool. Transactions combined with persistent process pairs are a solution. They provide simple process pairs (persistent execution) and a simple fault model.

In the limit, all faults are software faults—software is responsible for masking all the other faults. The best idea is to write perfect programs, but that seems infeasible. The next-best idea is to tolerate imperfect programs. The combination of failfast, transactions, and system pairs or process pairs seems to tolerate many transient software faults.

3.9 A Cautionary Tale—System Delusion

The focus here on system failures ignores the common end-to-end problem of *system delusion*: The computer system and its communications and terminals are up, but the output is untrustworthy or useless because the database is inconsistent with reality. System delusion generally results from *incorrect transactions* and from *unrecorded real-world activities.*

A specific example of system delusion may help in understanding the problem. Company X built an elaborate system to optimize and control the inventory of spare parts used for repairs. The application tracks many field locations, each with various types of inuse parts and spare parts. It also tracks a few central stock locations where parts ordered from manufacturers are stored until they are needed in the field. The stock locations also receive from field sites any unneeded spares and any defective parts that require repair. In principle, each field location does a transaction any time it uses a part or removes it from use, and a transaction each time it needs parts from central stock or returns parts to central stock. A central administrator runs transactions to release parts from central stock to the field, to order more parts from vendors, to record their receipt, and so on. As stated, this is a classic inventory-control problem.

As it turns out, the big problem with this system is not system failures. These do occur, but they are minor compared to the difficulties caused by unrecorded activities and by incorrect transactions.

Unrecorded real-world activities were a problem for various reasons. For example, optimal spare stocking levels are not intuitively obvious to people in the field. Most field managers gradually acquired extra spares, beyond what a reasonable inventory manager would agree was needed. Exactly how the field managers did this wasn't entirely clear, but they did it. Then, every once in a while they would get tired of seeing spares sitting around gathering dust and would dump the excess into cartons and send it back to central stock, but without any indication of where it was sent from (this to avoid admitting they had the stuff in the first place). Such activities guaranteed that the transaction system was constantly out of touch with reality, which made inventory management remarkably difficult. In particular, the spare reorder process tended to be seriously unstable.

This problem was eventually solved by adding heuristics to detect locations reporting activities inconsistent with recorded stocks. These tests triggered a physical reinventory of those sites. Even that was an art. For example, the field managers routinely hid excess inventory in the women's rest rooms.