```
proc prepareToCommit(tid)
    var t:  int # index into the status table

    # search transaction table for entry for tid.  let t be index of entry
        ...

    # write modified objects to the "non-current" copy
    fa i := 1 to statusTable[t].numDataItems ->
        ss.write(statusTable[t].dataAddrs[i] + (currentPointers mod 2 + 1),
                sizeof(data), statusTable[t].memCopy[i])
    af
    free(statusTable[t].memCopy)
end prepareToCommit


proc commit(tid)
    var t:  int # index into the status table

    # search transaction table for entry for tid, let t be index of entry
        ...
    # if entry cannot be found, return---transaction has committed already
        ...

    if statusTable[t].transStatus = 'A' -> # transaction hasn't committed yet
        # replace current pointers of data items by new current pointers
        fa i := 1 to statusTable[t].numDataItems ->
            ss.write(statusTable[t].dataAddrs[i], sizeof(int),
                (currentPointers mod (sizeof(data)+1) + 1))
        af
        statusTable[t].transStatus := 'D' # mark transaction as done
    fi
    if statusTable[t].transStatus = 'D' -> # cleanup
        ss.write(statusTable + t*sizeof(transInfoRec), sizeof(trans), statusTable[t])
        lockManager.unlock(tid, statusTable[t].dataAddrs)
        statusTable[t].transStatus := 'E' # mark table slot as being empty
        ss.write(statusTable + t*sizeof(transInfoRec), sizeof(trans), statusTable[t])
    fi
end commit
```

Figure 11: `prepareTocommit` and `commit` Operations

```
proc read(tid, dataAddrs, data, numDataItems);
    # search transaction table for entry for tid.  let t be index of entry
        ...

    fa i := 1 to numDataItems ->
        j = 1;
        do (statusTable[t].dataAddrs[j] != dataAddrs[i] ->
            j++
        od
        data[i] = statusTable[t].memCopy[j]
    af
end read
```

Figure 10: `read` Operation

items in the stable store to point to the new version written by `prepareToCommit`. Following this, the status of the transaction is changed to done ('D') in both the volatile and stable storage versions. The data items are then unlocked. Finally, the transaction status is changed to empty ('E'), with the change being reflected onto stable store as well. Since the transaction manager that co-ordinates the various data managers may re-issue commits when recovering from a failure, the `commit` operation may be re-executed in part or in total an arbitrary number of times given inopportune failures. Our implementation takes this into account by constructing this operation as a restartable action. The `abort` operation is similar to the `commit` and is therefore omitted for brevity.

Appendix B: Implementation of the Data Manager Resource

Here we present the rest of the procs exported by the data manager described in Section 4. As outlined there, the data manager keeps track of all in-progress transactions in a status table `transStatus`.

```
proc startTransaction(tid, dataAddrs, numDataItems)
    var t:  int # index into the status table

    P(statusTableMutex);
    # find an empty slot t in the statusTable
        ...
    statusTable[t].transStatus := 'A' # mark transaction as active

    statusTable[t].tid := tid V(statusTableMutex);
    # acquire locks on data items
    lockManager.lock(tid, dataAddrs)

    statusTable[t].memCopy := new(dataArray)
    statusTable[t].numItems := numDataItems
    fa i := 1 to numDataItems ->
        statusTable[t].dataAddrs[i] := dataAddrs[i]
        ss.read(objectAddrs[i], sizeof(int), currentPointer)
        statusTable[t].currentPointers[i] := currentPointer
        ss.read(dataAddrs[i]+currentPointer, sizeof(data), statusTable[t].memCopy[i])
    af

    # write status table entry onto stable store
    ss.write(statusTable + t*sizeof(transInfoRec), sizeof(trans), statusTable[t])
end startTransaction
```

Figure 9: `startTransaction` Operation

Figure 9 shows `startTransaction`. The proc first finds an empty slot in the `statusTable`, i.e. a slot with a `transStatus` of 'E', and marks it as being actively used ('A') by transaction `tid`. The lock manager is then invoked to acquire locks on the data items. The status table entry is updated next; specifically, the address of the data array is assigned to `memCopy` and the `numItems` field is initialized. Information concerning each data item is then stored after being retrieved from stable storage if necessary; this information includes the address of the data item, its value, and its `currentPointer`. Finally, the appropriate status table entry on stable storage is updated. Once a transaction is started, the data items it uses may be accessed and modified using the `read` and `write` operations. These operations use the copy of the data items in volatile memory. Figure 10 shows `read`; the `write` operation being very similar is not shown here.

The `prepareToCommit` and `commit` operations are shown in Figure 11. `prepareToCommit` is invoked when the transaction is ready to commit; it writes all the data items from the copy in volatile storage to the "non-current" copy in stable storage, and then discards the copies in volatile memory. The `commit` operation commits the modifications by changing the offset indicators of the appropriate data

```
resource buffer
    op fetch() returns value:  int
    op deposit(val newvalue:  int)
body buffer(size:  int)
    var first, last:  int := 0, 0
    var slot[0:size - 1]:  int

    initial
        send buff_loop()
    end

    proc buff_loop()
        do true ->
            in deposit(newvalue) and first != (last + 1) % size ->
                slot[last] := newvalue
                last := (last + 1) % size
            □ fetch() returns value and first != last ->
                value := slot[first]
                first := (first + 1) % size
            ni
        od
    end

end
```

Figure 8: Bounded Buffer

semantics, however, depending on how the proc is invoked (see below). The process terminates when (if) either its statement list terminates or a **return** is executed.

An operation can also be implemented as an alternative of an input statement. An input statement implementing a collection of operations $\texttt{opname}_1$, $\texttt{opname}_2$, ..., $\texttt{opname}_n$ has the following form:

```
in opname1(parameters) -> op_body1
□ opname2(parameters) -> op_body2
    ...
□ opnamen(parameters) -> op_bodyn
ni
```

A process executing an input statement is delayed until there is at least one alternative $\texttt{opname}_i$ for which there is a pending invocation. When this occurs, one such alternative is selected non-deterministically, the oldest pending invocation for the chosen alternative is selected, and the corresponding statement list is executed. The input statement terminates when the chosen alternative terminates.

An operation is invoked explicitly using a **call** or **send** statement, or is implicitly called by its appearance in an expression. The explicit invocation statements are written as

```
call op_denotation(arguments)
send op_denotation(arguments)
```

where the operation is denoted by a capability variable or by the operation name if the statement is in the operation's scope. An operation can be restricted to being invoked only by a call or a send by appending a {**call**} or {**send**} operation restrictor to the declaration of the operation.

Execution of a **call** terminates once the operation has been executed and a result, if any, returned. Its execution is thus synchronous with respect to the operation execution. Execution of a **send** statement is, on the other hand, asynchronous: a **send** terminates when the target process has been created (if a proc), or when the arguments have been queued for the process implementing the operation (if an input statement). Thus, the effects of executing the various combinations of **send/call** and **proc/in** are described by the following table.

Invocation	*Implementation*	*Effect*
call	**proc**	procedure call
send	**proc**	process creation
call	**in**	rendezvous
send	**in**	asynchronous message passing

To illustrate how the individual pieces of the language fit together, consider the implementation of a bounded buffer shown in Figure 8. Two operations are exported from this resource: `deposit` and `fetch`; `deposit` places a value in the next available slot if one exists, while `fetch` returns the oldest value from the buffer. A depositing process is delayed should the buffer be full. Similarly, a fetching process is delayed whenever the buffer is empty. Note also that the resource has a parameter `size`; its value determines the number of slots in the buffer. The use of resource parameters in this way allows instances to be created from the same pattern, yet still vary to a certain degree. Finally, note the single input statement to implement both the deposit and fetch operations, and the use of a send statement in the initialization code to initiate the main (parameterless) proc `buff_loop`. Creating a process in this manner is so common that the keyword **process** can be used instead of **proc** as an abbreviation for the **send** in the resource initialization code and corresponding **op** declaration.

Appendix A: The SR Distributed Programming Language

An SR program consists of one or more *resources*. These resources can be thought of as patterns from which resource instances are created dynamically. Each resource is composed of two parts: an interface portion which specifies the interface of the resource and a *body*, which contains the code to implement the abstract object. The specification portion contains descriptions of objects that are to be exported from this resource—made available for use within other resources—as well as the names of resources whose objects are to be imported. Of primary importance are the declaration of *operations*—actions implemented by sequences of statements that can be invoked. These declarations specify the interface of those operations that are available for invocation from other resources. For example,

```
op example1(var x: int; val y: bool)
```

declares an operation, `example1`, that takes as arguments an integer `x` that is passed with copy-in/copy-out (**var**) semantics and a boolean `y` that is copy-in only (**val**). Result parameters (**res**) are also supported, as are operations with return values.

The declaration section in the resource body together with its specification define the objects that are global to the resource, i.e., accessible to any process within the resource. All of the usual types and constructors are provided. In addition, there are *capability variables*. Such a variable functions either as a pointer to all operations in a resource instance (a *resource capability*), or as a pointer to a specific operation within an instance (an *operation capability*). A variable declared as a resource capability is given a value when a resource instance is created, while an operation capability is given a value by assigning it the name of an operation or from another capability variable. Once it has a value, such variables can be used to invoke referenced operation(s), as described later.

The resource instances comprising a given program may be distributed over multiple *virtual machines*, which are abstract processors that are mapped to physical machines in the network. A resource instance is created and placed on a virtual machine using the following:

```
res_cap := create res_name(arguments) on virtual_machine_cap
```

Execution of this statement creates an instance of the resource `res_name` on the virtual machine specified by the virtual machine capability `virtual_machine_cap` and assigns a capability to the newly created resource to the capability variable `res_cap`.

An operation is an entry into a resource. An SR operation has a name, and can have parameters and return a result. There are two different ways to implement an operation: as a *proc* or as an alternative in an *input* statement. A proc is a section of code whose format resembles that of a conventional procedure:

```
proc opname(parameters) returns result
    op_body
end
```

The operation body `op_body` consists of declarations and statements. Like a procedure, the declarations define objects that are local to the operation `opname`. Unlike a procedure, though, a new process is created, at least conceptually, each time `opname` is invoked. It is possible to get standard procedure-like

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[LW85] R. J. LeBlanc and C. T. Wilkes. Systems programming with objects and actions. In *The 5th International Conference on Distributed Computing Systems*, pages 132–139, Denver, Colorado, May 1985. IEEE Computer Society.

[MSMA90] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, January 1990.

[PHOR90] L. L. Peterson, N. C. Hutchinson, S. W. O'Malley, and H. C. Rao. The *x*-Kernel: A platform for accessing internet resources. *IEEE Computer*, 23(5):23–33, May 1990.

[PS88] F. Panzieri and S. K. Shrivastava. Rajdoot: A remote procedure call mechanism supporting orphan detection and killing. *IEEE Transactions on Software Engineering*, 14(1):30–37, January 1988.

[PVB+88] D. Powell, P. Verissimo, G. Bonn, F. Waeselynck, and D. Seaton. The Delta-4 approach to dependability in open distributed computing systems. In *Digest of Papers, The Eighteenth International Symposium on Fault-Tolerant Computing*, pages 246–251, June 1988.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[SCP91] R. D. Schlichting, F. Cristian, and T. D. M. Purdin. A linguistic approach to failure-handling in distributed systems. In Algirdas Avižienis and Jean-Claude Laprie, editors, *Dependable Computing for Critical Applications*, volume 4 of *Dependable Computing and Fault-Tolerant Systems*, pages 387–409. Springer-Verlag, Wien and New York, 1991.

[SS83] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *IEEE Transactions on Computing Systems*, 1(3):222–238, August 1983.

[Wei89] W. E. Weihl. Using transactions in distributed applications. In Sape Mullender, editor, *Distributed Systems*, pages 215–235. Addison-Wesley Publishing Company, ACM Press, New York, New York, 1989.

[CAS85] F. Cristian, H. Aghili, and R. Strong. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Digest of Papers, The Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206. IEEE Computer Society, June 1985.

[CGR86] R. F. Cmelik, N. H. Gehani, and W. D. Roome. Fault tolerant concurrent C: A tool for writing fault tolerant distributed programs. In *Digest of Papers, The Eighteenth International Symposium on Fault-Tolerant Computing*, pages 55–61. IEEE Computer Society, IEEE Computer Society Press, June 1986.

[CM84] J. Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[Coo85] E. C. Cooper. Replicated distributed programs. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 63–78. ACM SIGOPS, December 1985.

[Cri91] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[Dim85] C. I. Dimmer. The Tandem Non-Stop system. In T. Anderson, editor, *Resilient Computing Systems*, chapter 10, pages 178–196. John Wiley & Sons, 1985.

[DoD83] U.S. Department of Defence. *Reference Manual for the Ada Programming Language*. Washington D.C., 1983.

[EFH82] C.S. Ellis, J.A. Feldman, and J.E. Heliotis. Language constructs and support systems for distributed computing. In *ACM Symposium on Principles of Distributed Computing*, pages 1–9. ACM SIGACT-SIGOPS, August 1982.

[GMS89] H. Garcia-Molina and A. Spauster. Message ordering in a multicast environment. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 354–361, June 1989.

[Gra86] J. N. Gray. An approach to decentralized computer systems. *IEEE Transactions on Software Engineering*, SE-12(6):684–692, June 1986.

[HMPT89] N. C. Hutchinson, S. Mishra, L. L. Peterson, and V. T. Thomas. Tools for implementing network protocols. *Software — Practice and Experience*, 19(9):895–916, Sep 1989.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[HW87] M. P. Herlihy and J. M. Wing. Avalon: Language support for reliable distributed systems. In *Digest of Papers, The Seventeenth International Symposium on Fault-Tolerant Computing*, pages 89–94. IEEE Computer Society, IEEE Computer Society Press, July 1987.

[KTHB89] M. F. Kaashoek, A. S. Tanenbaum, S. Flynn Hummel, and H. E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

[KU87] J. C. Knight and J. I. A. Urquhart. On the implementation and use of Ada on fault-tolerant distributed systems. *IEEE Transactions on Software Engineering*, SE-13(5):553–563, May 1987.

[Lam81] B. W. Lampson. Atomic transactions. In B.W. Lampson, M. Paul, and H.J. Seigert, editors, *Distributed Systems—Architecture and Implementation*, chapter 11, pages 246–265. Springer-Verlag, 1981. Originally vol. 105 of Lecture Notes in Computer Science.

[Lap91] J.-C. Laprie. *Dependability: Basic Concepts and Terminology*, volume 4 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, 1991.

[Lis85] B. Liskov. The Argus language and system. In M. Paul and H.J. Siegert, editors, *Distributed Systems: Methods and Tools for Specification, Lecture Notes in Computer Science, Volume 190*, chapter 7, pages 343–430. Springer-Verlag, Berlin, 1985.

Systems like Rajdoot [PS88] provide some support for restartable actions in the form of orphan detection and termination, allowing the programmer to re-issue invocations to servers. These systems, however, make no provision for replication.

We feel that FT-SR provides an appropriate level of abstractions for a systems programming language. These abstractions are primitive enough to give the programmer the ability to build all other abstractions, yet powerful enough to be able to do so with relative ease. Such flexibility allows FT-SR to be used for a variety of different applications and system architectures. We also expect that the primitive nature of these abstractions will allow them to be efficiently implemented, an important consideration for a systems programming language.

7 Concluding Remarks

A distributed programming language designed to support the construction of fault-tolerant systems must be flexible enough to allow a variety of structuring and redundancy techniques. FT-SR has been designed to be such a language by incorporating facilities targeted at supporting the various programming paradigms that have been proposed for such systems. These include support for encapsulation based on SR resources, synchronous and asynchronous failure notification, resource replication with consistent invocation ordering, and recovery. The logical basis of the language design is a programming model centered around the notion of fail-stop atomic objects.

Future work will concentrate on the task of completing the implementation and using the language to construct a number of different prototype systems. This process will be used to gain experience with the language that can be used to refine and expand the design. Among the additional issues that we expect to address are the expansion of our failure-handling mechanisms into a general exception handling scheme oriented towards the specific nature and requirements of distributed programming languages, and the incorporation of provisions for real-time computing.

Acknowledgements

We would like to thank Greg Andrews, Ron Olsson, Henri Bal, and Matti Hiltunen for reading an earlier version of this paper and providing us with valuable feedback.

References

[AOC+88] G. R. Andrews, R. A. Olsson, M. A. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[Bal91] H. E. Bal. A comparative study of five parallel programming languages. In *Proceedings of EurOpen Conference on Open Distributed Systems*, May 1991.

[BH75] P. Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE-1(2):199–206, June 1975.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, 1987.

[BSS91] K. Birman, A. Schipe, and P. Stephenson. Lightweight causal and atomic group multicast. Technical Report 91-1192, Department of Computer Science, Cornell University, February 1991.

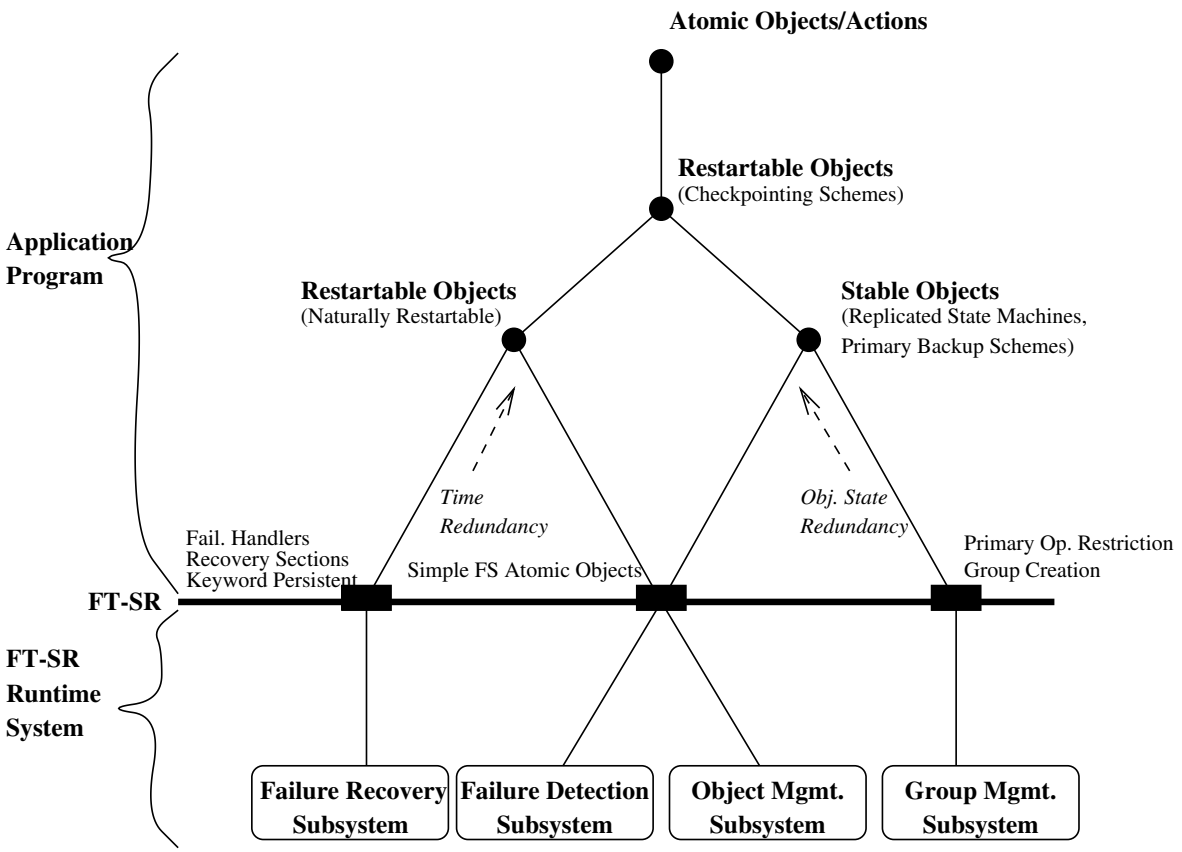

Figure 7: Abstraction Hierarchy

fault-tolerant programming. In particular, FT-SR provides the abstraction of a simple FS atomic object and two categories of language mechanisms: those that deal with failure recovery and those that deal with group management. A simple FS atomic object may be combined with these language mechanisms in a variety of ways to realize the abstractions provided by the different programming paradigms. For example, as illustrated in our banking example, an object with a recovery section can be used to implement a restartable object. Similarly, the FT-SR create statement can be used to group objects to form a stable object such as stable storage. In addition, FT-SR gives the programmer the ability to create groups where the degree of replication is automatically maintained and the ability to restart entire groups after a catastrophic failure.

Other programming languages that have been proposed for fault-tolerant programming differ from FT-SR in that they provide support for only a limited number of these abstractions. Argus [Lis85], for instance, provides its users with the atomic object/action abstraction. The Argus language runtime system therefore has to implement all the other underlying abstractions except stable storage, which it assumes is provided by the hardware. Languages with checkpointing primitives also assume the existence of stable storage and implement in the language runtime the mechanics of checkpointing. As another example, languages like FTCC [CGR86] provide the programmer with the ability to create and manage object groups and, as a result, the ability to build stable objects, but lack support for building restartable objects.

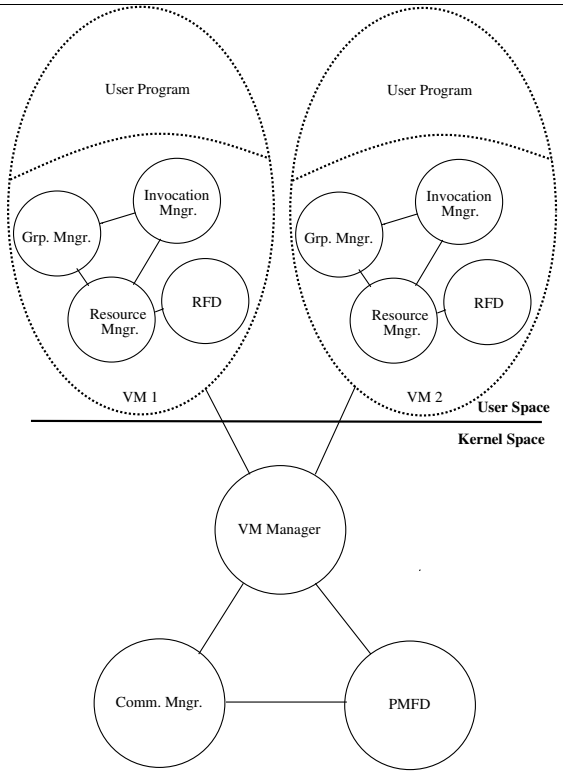

Figure 6: Organization of the FT-SR Run-Time System

manage intra-virtual machine communication. The portion of the runtime system that manages virtual machines and inter-virtual machine communication resides inside the kernel. Among the most significant modules in the kernel space are the Communication Manager, the Virtual Machine Manager, and the VMFD. The Resource Manager, the Group Manager, the Invocation Manager, and the RFD reside in each of the virtual machines and form the bulk of the runtime system. The communication paths between these modules are also shown in Figure 6.

6 Discussion

The various programming paradigms that have been developed for fault-tolerant distributed systems provide the programmer with system structuring techniques and abstractions for the problem being solved. The relationship between these abstractions can be illustrated by arranging them in a hierarchy, such that every member of the hierarchy is built using the abstractions directly below it. This hierarchy is shown in Figure 7, where the circles represent abstractions defined by the programming paradigms, the rectangles represent the abstractions and mechanisms provided by FT-SR, and the labelled boxes at the bottom represent the portion of the FT-SR runtime system that implement the language's abstractions and mechanisms.

From this figure, it is clear that FT-SR does indeed provide the fundamental abstractions that underlie

has been made to keep the implementation as efficient as possible. For example, whenever possible, the implementation takes advantage of the fact that the processors can only suffer from fail-silent failures and that the maximum number of simultaneous failures *max_sf* is known *a priori*.

Failure Detection and Notification. Failure detection and notification is the single most important difference between FS atomic objects and SR resources. Failure detection is initiated in one of two ways: when a resource explicitly asks to be notified of a failure using the **monitor** statement, or when the communication module of the runtime system cannot complete an invocation and suspects a failure. Depending on how the failure detection was initiated, the runtime system either notifies the user program of the failure by generating an implicit invocation of the operation specified by the **monitor** statement, or, if a backup operation has been supplied with an invocation, the invocation is forwarded to the backup operation.

Failure detection in FT-SR is done at two levels: at the virtual machine level and at the resource level. Every processor in the distributed system has a physical machine failure detector (PMFD) that monitors the physical machines in the system and notifies the VM Manager of any failures. The VM Manager monitors the virtual machines on the processor on which it is executing for termination, and can, with the cooperation of other VM Managers and the failure notifications from the PMFD, detect the failure of any virtual machine on the network due to processor crashes. Resource level failure detection is done at each virtual machine by a resource failure detector or RFD. An RFD is analogous to the VMFD in function: it monitors resources on the virtual machine and, with the help of other RFDs on other virtual machines, can detect the failure of any resource in the system.

Replication and Recovery. FT-SR provides two mechanisms for increasing failure resilience: replication and recovery. For replication, the most interesting aspect of the implementation is managing group communication, since messages sent to the group as a result of invocations have to be multicast and delivered in a consistent total order. The technique we use is similar to [CM84, KTHB89, GMS89], where one of the replicas is a primary through which all messages are funneled. Upon receiving a message, the primary adds a sequence number and multicasts it to the members of the group. As soon as the primary gets *max_sf* acknowledgements, it sends an acknowledgment to the original sender of the message; this action is appropriate since the receipt of *max_sf* acknowledgements guarantees that the message will not be lost even with multiple failures. The primary is also involved when messages are sent by the group as a whole. The runtime system suppresses group invocations from all group members except the primary. When the primary receives acknowledgements for these invocations it multicasts them to the group members.

Supporting recovery involves (1) restarting the resource instance, either as a result of an explicit request or due to its declaration as persistent, (2) ensuring that the recovery code is executed, and (3) correctly starting the delivery of new invocations. Actually implementing (1) and (2) are fairly easy since the requirements for restarting a resource instance are very similar to creating one initially. For a persistent resource, this is preceded by the selection of a backup virtual machine from the list supplied during initial creation to act as the new host.

Figure 6 shows the organization of the FT-SR runtime system on a single processor. The implementation uses the *x*-kernel [HMPT89, PHOR90], a stand-alone operating system kernel that provides memory and process management, as well as a flexible infrastructure for constructing communication protocols. Each FT-SR virtual machine exists in an *x*-kernel address space and contains those parts of the runtime system that create and destroy resources, route invocations to operations on resources, and

```
resource main
    imports transManager, dataManager, stableStore, lockManager
body main
    var virtMachines[4] :  cap vm # array of virtual machine capabilities
    dataSS[2], tmSS: cap stableStore # capabilities to stable stores
    lm:  cap lockManager; dm[2]:  cap dataManager # capabilities to lock and data managers

    virtMachines[1] := create vm() on ``host1''
    virtMachines[2] := create vm() on ``host2''
    virtMachines[3] := create vm() on ``host3'' # backup machine
    virtMachines[4] := null

    # create stable storage for use by the data managers and the transaction manager
    dataSS[1] := create (i := 1 to 2) stableStore() on virtMachines
    dataSS[2] := create(i := 1 to 2) stableStore() on virtMachines
    tmSS := create (i := 1 to 2) stableStore() on virtMachines

    # create lock manager on ``host2''
    lm := create lockManager() on virtMachines[2]

    # create data managers
    fa i := 1 to 2 ->
        dm[i] = create dataManager(i, lm, ss1) on virtMachines[i]
    af

    # create transaction manager
    tm = create transManager(dm[1], dm[2], ss3) on virtMachines[1]
end main
```

Figure 5: System startup in Resource `main`

The main resource that starts up the entire system is shown in Figure 5. Resource `main` creates a virtual machine on each of the three physical machines available in the system. Three stable storage objects are then created, where each such object has two replicas and uses the virtual machine on "host3" as a backup machine. The two data managers are then created followed by the transaction manager. Notice how the system is created "bottom up," with the objects at the bottom of the dependency graph being created before the objects on which they depend. This way, each object can be given capabilities to the objects on which it depends upon creation.

5 Implementation Strategy

There are two major aspects in which FT-SR differs from SR: the support for using resources as FS objects, and the support for techniques that enhance failure resilience. Accordingly, we focus in this section on describing those parts of the FT-SR language runtime system that provide this support. It is worth keeping in mind that, since FT-SR is designed to be a systems programming language, every effort

they can be found in Appendix B.

We now turn to implementing the stable storage assumed in the above. One way of realizing this abstraction is by using the state machine approach, that is, by creating a storage resource and replicating it to increase failure resilience. Figure 4 shows such a stable store resource; for simplicity, we assume that storage is managed as an array of bytes.

```
persistent resource stableStore
    uses globalDefs
    op read(address: int; numBytes: int; buffer: charArray)
    op write(address: int; numBytes: int; buffer: charArray)

    op sendState(sscap: cap stableStore)
    op recvState(objectStore: objList)
body stableStore
    var store[MEMSIZE]: char

    process ss
        in read(address, numBytes, buffer) -> copy(buffer, store[address], numBytes)
        □ write(address, numBytes, buffer) -> copy(store[address], buffer, numBytes)
        □ sendState(rescap) -> send rescap.recvState(store)
        ni
    end ss

    recovery
        send mygroup().sendState(myresource())
        receive recvState(store)
        send ss
    end recovery
end stableStore
```

Figure 4: `stableStore` Resource

Replica failures are dealt with by restarting the resource on another machine; this is done automatically since `stableStore` is declared to be a persistent resource. The recovery code that gets executed in this scenario starts by requesting the current state of the store from the other group members. All replicas respond to this request by sending a copy of their storage state; the first response is received, while the other responses remain queued at the `recvState` operation until the replica is either destroyed or fails. The newly restarted replica begins processing queued messages when it is finished with recovery. Since messages are queued from the point that its `sendState` message was sent to the group, the replica can apply these subsequent messages to the state it receives to reestablish consistency with the state of the other replicas.

The stable store may also be implemented as a primary-backup group by adding a {**primary**} restriction to the `read` and `write` operations. The process `ss` would then send the updated state to the rest of the group at the end of each operation by invoking a `recvState` operation on the group. This operation would be implemented by extending the input statement in `ss` to include this operation as an additional alternative.

```
body dataManager
    type transInfoRec = rec(tid:  int; transStatus:  int
            dataAddrs:  addressList #address of data in stable store
            currentPointers:  intArray
            memCopy:  ptr dataArray #pointer to in-core copy of data
            numItems:  int)
    var statusTable[1:MAX_TRANS]: transInfoRec; statusTableMutex:  semaphore

    initial
        # initialize statusTable
            ...
        monitor(ss)send failHandler()
        monitor(lmcap)send failHandler()
    end initial

    proc startTransaction(tid, dataAddrs, numDataItems)
        ...code for startTransaction...
    end startTransaction

    proc prepareToCommit(tid)
        ...code for prepareToCommit...
    end prepareToCommit

    proc commit(tid)
        ...code for commit...
    end commit

    proc abort(tid)
        ...code for abort...
    end abort

    proc read/write(tid, dataAddrs, data)
        ...code for read/write...
    end abort

    proc failHandler()
        ss.read(statusTable, sizeof(statusTable), statusTable);
        transManager.dmUp(dmId);
    end failHandler

    recovery
        destroy myresource()
    end recovery
end dataManager
```

Figure 3: Outline of the Data Manager Resource

```
resource dataManager
    uses globalDefs
    imports lockManager, stableStore
    op startTransaction(tid: int; dataAddrs: addrList; numDataItems: int)
    op read(tid: int; dataAddrs: addrList; data: dataList; numDataItems: int)
    op write(tid: int; dataAddrs: addressList; data: dataList; numDataItems: int)
    op prepareToCommit(tid: int), commit(tid: int), abort(tid: int)
body dataManager(dmId: int; lmcap: cap lockManager; ss: cap stableStore) separate
```

Figure 2: Specification for resource `dataManager`

implemented as procs; thus, invocations result in the creation of a thread that executes the statements in the proc concurrently with other threads. Finally, the data manager contains initial and recovery code, as well as a failure handler proc that deals with the failure of the `lockManager` and `stableStore` resources.

To implement the atomic commitment of the transaction, the data manager utilizes the standard technique of maintaining two versions of each data item together with an indicator of which is current [BHG87]. To simplify our implementation, we maintain this indicator and the two versions in contiguous memory locations, with the indicator being an offset and the address of the indicator used as the logical address of the item. Thus, the actual address of the current copy of the item is calculated by taking the address of the item and adding to it the indicator offset.

The data manager keeps track of all in-progress transactions in a *status table*. This table contains for each active transaction the `tid`, the status (`transStatus`), the stable storage addresses of the data items being accessed by the transaction (`dataAddrs`), the value of the indicator offset of each item (`currentPointers`), a pointer to an array in volatile memory containing a copy of the data items (`memCopy`), and the number of data items being used in the transaction (`numItems`). This table can be accessed concurrently by threads executing the procs in the body of the data manager, so the semaphore `statusTableMutex` is used to achieve mutual exclusion. New entries in this table also get saved on stable storage for recovery purposes. Reads and writes during execution of the transaction are actually performed by the data manager on versions of the items that it has cached in its local (volatile) storage.

The data manager depends on the stable storage and lock manager to correctly implement its operations. As a result, it needs to be informed when they fail in such a way that the abstraction they implement is destroyed. The data manager does this by establishing an asynchronous failure handler `failHandler` for both of these events in the initial code by using the monitor statement. When invoked, `failHandler` terminates the data manager resource, thereby causing the failure to be propagated to the transaction manager.

The failure of the data manager itself is handled by recovery code that retrieves the current contents of the status table from the stable store. It is the responsibility of the transaction manager to deal with transactions that were in progress at the time of the failure; those for which `commit` had not yet been invoked are aborted, while `commit` is reissued for the others. To handle this, the recovery code sends a message to the transaction manager notifying it of the recovery.

The procs implementing the other data manager operations do not use any of the FT-SR primitives specifically designed for fault-tolerant programming and are therefore not shown here. For completeness,

a new instance. This means, for example, that other resource instances can invoke its operations using any capability values for it that they possessed prior to the failure.

Implicit restarting of failed resource instances is also supported. This is designed to allow a resource group to automatically regain its original level of redundancy following a failure. Automatic restart is indicated by the presence of the keyword **persistent** on the resource declaration and the inclusion of more virtual machines in the `vm_list` of the original create statement than the number of replicas actually created. Then, should a virtual machine executing one of the instances of the resource group fail, the system will select one of these *backup virtual machines* and recreate the failed instance automatically. The arguments supplied during the recreation are the same as those used for the original creation.

The one remaining issue concerning restart is determining when the runtime of the recovering resource instance begins accepting invocations from other instances. In general, the resource is in an indeterminate state while performing recovery, so we choose to begin accepting messages only after the recovery code has completed. The one exception to this is if the recovering instance itself initiates an invocation during recovery; in this case, invocations are accepted starting at the point that invocation terminates. This is to facilitate a system organization in which the recovering instance retrieves state variables from other resources during recovery.

4 Programming with FT-SR

In this section, we present an example program that illustrates how FT-SR can be used to program a simple fault-tolerant system that uses multiple programming paradigms. The example consists of the data manager and stable storage objects from the banking system described in Section 2. As outlined there, the data manager implements a collection of operations that provide transactional access to data items located on a stable store. The organization of the manager itself is based on the restartable action paradigm, with key items in the internal state being saved on stable storage for later recovery in the event of failure. After describing portions of the data manager, we then show how stable storage can be built by replicating a storage resource, with the replicas being configured using either the replicated state machine approach or a primary-backup scheme.

The data manager controls concurrent accesses and provides atomicity for access to data items on stable storage. For simplicity, we assume that all data items are of the same type and are referred to by a logical address. The stable store is read by invoking its read operation, which takes as arguments the address of the block to be read, the number of bytes to be read, and a buffer in which the values read are to be returned. Data is written to the stable store by invoking an analogous write operation, which takes as arguments the starting address of the block being written, the number of bytes in the block, and a buffer containing the values to be written.

Figures 2 and 3 show the specification and an outline of the body of such a data manager. As can be seen in its specification, the data manager imports a stable store and lock manager resource and exports six operations: `startTransaction`, `read`, `write`, `prepareToCommit`, `commit`, and `abort`. The operation `startTransaction` is invoked by the transaction manager to access data held by the data manager; its arguments are a transaction identifier `tid` and a list of addresses of the data items used during the transaction. `read` and `write` are used to access and modify objects. The two operations `prepareToCommit` and `commit` are invoked in succession upon completion to, first, commit any modifications made to the data items by the transaction, and, second, terminate the transaction. `abort` is used to abandon any modifications and terminate the transaction; it can be invoked at any time up to the time `commit` is first invoked. Note that all of these operations exported by the data manager are

atomically, so that either all replicas receive the invocation or none do. This property is guaranteed by the runtime given no greater than *max_sf* simultaneous failures, where *max_sf* is a parameter set by the user at compile time. The combination of the atomicity and consistent ordering properties means that an invocation using a resource group capability is equivalent to an *atomic broadcast* [CAS85, MSMA90].

In addition to this facility of dealing with invocations coming into a resource group, provisions are also made for coordinating invocations generated within the group. There are two kinds of invocations that can be generated by a group member. In some cases, a group member may wish to communicate with a resource instance as an individual even though it happens to be in a group. For example, this would be the situation if each replica has its own set of private resources with which it communicates. At other times, the group members might want to cooperate to generate a single outgoing invocation on behalf of the entire group. To distinguish between these two kinds of communication, FT-SR provides an **owns** clause that can be used in a resource specification to specify private resources. The **owns** clause is similar to the **imports** clause and is meaningful only in the context of replicated groups (when used by a non-replicated resource, **owns** is identical to **imports**). All invocations from a group member to its "owned" resources are considered private communication and not co-ordinated with other invocations from group members. However, invocations to "imported" resources are considered to be invocations from the entire group, so exactly one invocation is generated in this case. The invocation is actually transmitted when one of the group members reaches the statement, with later instances of the same invocation being suppressed by the language runtime system. Note, however, that this invocation could, in fact, be a multicast-type invocation as described above if the operation being invoked is within another resource group (i.e., if the capability used in the statement is a resource group capability).

In an analogous manner, if an "owned" resource is created by a member of a group, the resource instance created is completely independent of any created by other replicas. On the other hand, the creation of an "imported" resource causes only one resource to be created with all replicas getting the same capability as the return value.

A resource group can also be configured to work according to a primary-backup scheme [Dim85]. In this scenario, invocations to the group are delivered only to a replica designated as the primary by the language runtime, with the other replicas being passive. This type of configuration is achieved by placing the op restrictor {**primary**} on the declaration of operations in the group members that are to be invoked only if the replica is the primary.

FT-SR also provides the programmer with the ability to restart resource instances that were executing on failed virtual machines. The recovery code to be executed upon restart is denoted by placing it between the keywords **recovery** and **end** in the resource text. This syntax is analogous to the provisions for initialization and finalization code in the standard version of SR. A resource instance may be restarted either explicitly or implicitly. Explicitly, it is done by the following statement:

```
restart rescap(args) on vm_list
```

This restarts the resource indicated by the capability `rescap` and executes the specified recovery code; if no recovery code has been given, a run-time error is raised. To restart an entire resource group,

```
restart (i:=1 to N) groupcap(args) on vm_list
```

is used. The size of the reconstituted group can be different from the original. In both cases, it is important to note that the restarted resource instance is, in fact, a *recreation* of the failed instance and not

invocation if the resource implementing the operation fails. Both call and send invocations are guaranteed to succeed in the absence of failures.

FT-SR also provides the programmer with asynchronous failure notification, which can be requested by using the **monitor** statement. The statement

```
monitor(res_cap) send fail_handler(args)
```

enables monitoring of the resource instance specified by the resource capability `res_cap`. If that resource should subsequently fail, the operation `fail_handler` will be implicitly invoked with the specified arguments by the language runtime system. The expressions provided for argument values are evaluated at the time the monitor statement is executed and not when the failure occurs. Monitoring can be terminated by the **monitorend** function, which also takes a resource capability as its argument or by another monitor statement that specifies the same resource. The ability to request asynchronous notification has proven to be convenient and is in keeping with the inherently asynchronous nature of failures themselves [SCP91].

3.2 Increasing Failure Resilience

FT-SR provides mechanisms for supporting the construction of more resilient, higher-level FS atomic objects using replication, and for increasing the resilience of objects using recovery techniques. The replication facilities allow multiple copies of an FT-SR resource to be created, with the language and runtime providing the illusion that the collection is a single resource instance exporting the same set of operations. The SR create statement has been generalized to allow for the creation of such replicated resources, which we call a *resource group*. The statement

```
rescap := create (i := 1 to N) res(args) on vm_caps
```

creates a resource group with N identical instances of the resource `res` on the virtual machines specified by the array of virtual machine capabilities `vm_caps`. Both the quantifier `(i := 1 to N)` and **on** clauses are optional. If the quantifier is omitted, the statement reverts to the semantics of the normal SR statement, which creates one instance of the named resource; if the **on** clause is omitted, all of the instances are created on the current virtual machine.[1]

The value returned from executing the create statement is a resource capability that provides access to the operations implemented by the new resource(s). If a single resource instance is created, the capability allows the holder to invoke any of the exported operations in that instance as provided for in normal SR. If, on the other hand, multiple identical instances are created, the capability is a *resource group capability* that allows multicast invocation of any of the group's exported operations. In other words, using this capability in a **call** or a **send** statement causes the invocation to be multicast to each of the individual resource instances that make up the group. All such invocations are guaranteed to be delivered to the runtime of each instance in a consistent total order, although the program may vary this if desired. This means, for example, that if two operations implemented by alternatives of an input statement are enabled simultaneously, the order in which they will be executed is consistent across all functioning replicas unless a scheduling expression by the programmer overrides this explicitly. Moreover, the multicast is also done

[1]This is the natural generalization of the standard SR semantics, although obviously not especially useful for increasing failure resilience to processor crashes.

3 FT-SR Language Description

The goal of FT-SR is to support the building of systems based on the FS atomic object model, and thus, by implication, the building of systems using any of the existing programming paradigms. Given the need for flexibility, we do not provide these objects directly in the language, but rather include features that allow them to be easily implemented. To this end, the language has provisions for encapsulation based on SR resources, resource replication, recovery protocols, synchronous failure notification when performing interprocess communication, and a mechanism for asynchronous failure notification based on a previous scheme for SR [SCP91]. Since our extensions are based on existing SR mechanisms, a short overview of the language is provided in Appendix A; for further details, see [AOC+88].

3.1 Simple FS Atomic Objects

Realizing much of the functionality of a simple FS atomic object—i.e., one not composed of other objects or using any other failure resilience techniques—in SR is straightforward since a resource instance is essentially an object in its own right. For example, it has the appropriate encapsulation properties, and is populated by a varying number of processes that can function as threads in the FS atomic object model. SR operations are also very similar to the operations defined by the model; they are implemented by processes and can be exported for invocation by processes in other resource instances. Moreover, the execution semantics of SR operations are already close to those desired for FS atomic objects; the only additional property required is atomicity in the absence of failures, that is, atomicity with respect to concurrent execution. This can easily be programmed in SR by, for example, implementing each exported operation as a separate alternative in an input statement repeatedly executed by a single process. Standard locking-based solutions that allow more concurrency while maintaining the semantics of atomicity are also easy to implement in SR.

The one aspect of simple FS atomic objects that SR does not support directly—and hence the focus of our extensions in this area—is generation of a failure notification upon failure of the abstraction. For simple objects, this occurs when the virtual machine on which the resource instance is executing is lost due to a processor failure or network partition, or if it is explicitly destroyed from within the program. In Section 5, we discuss how this failure is detected by the language runtime code, so here we concentrate on describing the mechanisms that are provided to field this notification in other resource instances. These facilities allow an abstract object to react to the failure of other objects on which it depends.

FT-SR provides the programmer with two different kinds of failure notification and consequently, two different ways of fielding a notification. The first kind is notification that is synchronous with respect to a call to a resource; it is fielded by an optional backup operation specified in the calling statement. The following program fragment illustrates the use of such a backup operation:

```
call <operation, backupOp>(arg1, arg2, ..., argN)
```

The operation `backupOp` is invoked should the call to `operation` fail, where a call is defined to fail if the resource instance implementing the operation being invoked fails before it can reply to the call. The backup operation is called with the same arguments as the original operation and, hence, must be type compatible with the original operation. Execution is blocked if a call fails and there is no associated backup operation. Note that such backup operations can only be specified with call invocations; send invocations are non-blocking and no guarantees can be made about the success or failure of such an

failures of objects upon which they depend. If such a failure cannot be tolerated, it may, in turn, cause subsequent failures to be propagated up the dependency graph. At the top level, this would be viewed by the user as the catastrophic failure of the transaction manager and hence, the system. Such a situation might occur, for example, should the redundancy being used to implement stable storage be exhausted by an untimely series of failures. In Section 4, we illustrate how the data manager and stable storage FS atomic objects from this example might be implemented using FT-SR.

Fail-stop atomic objects and the associated techniques for increasing failure resilience form, in our view, a "lowest common denominator" that can be conveniently used to realize the seemingly disparate programming paradigms proposed for fault-tolerant programming. For example, consider the object/action model. In this model, there are two types of components, objects and actions. Objects are passive entities that export operations, whereas actions are active entities akin to threads that invoke a series of operations from potentially different objects to carry out their tasks. An action has the following properties related to execution atomicity [Lam81]. First, it is *unitary*, that is, it is either executed completely or not at all, despite failures; second, it is *serializable*, that is, the effect of executing multiple actions concurrently is equivalent to some serial schedule. These properties have also been called totality and serializability [Wei89], and recoverability and indivisibility [Lis85]. In the database literature, atomic actions are known as transactions [BHG87].

A system using the atomic object/action paradigm may be implemented using FS atomic objects. Objects in the system correspond to FS atomic objects. An action corresponds to an abstract thread realized by the combination of concrete threads in the FS atomic objects. This abstract thread may span multiple FS atomic objects as a result of invocations made by concrete threads that are serviced by concrete threads in other objects. Standard locking and commit protocols are used to ensure the unitary and serializable nature of these actions across multiple objects. Viewed as a whole, this system appears to the user as one FS atomic object exporting the set of operations required by the application.

As a second example, consider the replicated state machine paradigm. In this paradigm, a system is implemented as a collection of interacting state machines. Each such machine consists of *state variables*, which encode its state, and *commands*, which transform its state. Each command is implemented by a deterministic program that modifies the state variables and/or produces some output. Command executions are atomic and any outputs produced are determined solely by the sequence of commands processed by the state machine. A fault-tolerant version of a state machine can be implemented by replicating that state machine and running each replica on a different processor in a distributed system. All available replicas must receive all commands sent to the replicated state machine (the *agreement* property) and must process them in the same sequence (the *order* property).

State machines map directly to FS atomic objects. Commands to the state machine correspond to invocations on an FS atomic object, with locking techniques being used to ensure that the operation executions are atomic. A replicated state machine can be implemented by replicating FS atomic objects and ensuring that all commands to the state machine result in invocations on all replicas in a consistent order. Each ensemble of replicated FS atomic objects forms a higher-level FS atomic object representing the fault-tolerant version of a given state machine. The entire collection of such FS atomic objects can then be viewed as a single FS atomic object that implements the entire system.

The system shown in Figure 1 is an example of a system in which FS atomic objects are used to implement different programming paradigms in different parts of the system. Specifically, the transaction and data managers are built using the restartable action paradigm, while the stable storage objects are built using the replicated state machine approach. The user of the banking system sees the system as one implementing the atomic object/action paradigm and interacts with it accordingly.

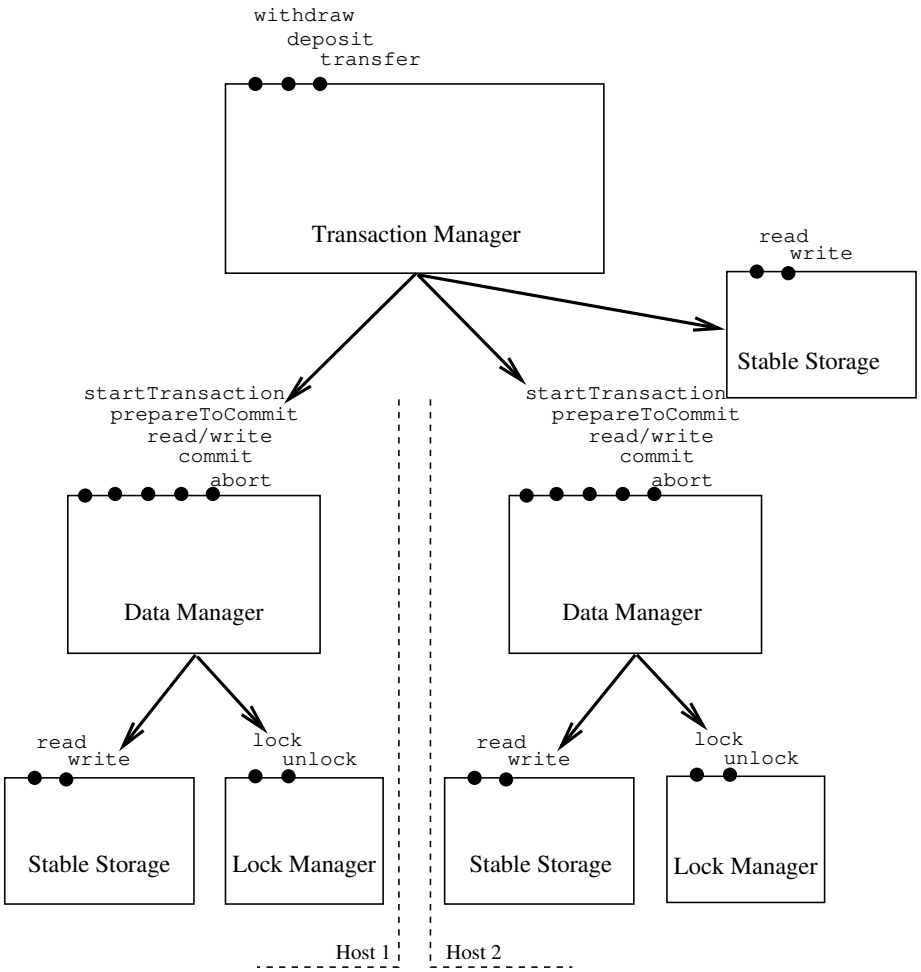

Figure 1: Fault-tolerant system structured using FS atomic objects

its machine.

The user interacts with the transaction manager, which in turn uses the data managers and a stable storage object to implement transactions. The role of the transaction manager is to be the coordinator, deciding if and when a transaction is to be committed; it uses the stable storage object to log the progress of transactions in the system. The data managers export operations that are used by the transaction manager to deal with transactions on that host. The stable storage associated with each data manager is used to store the actual data corresponding to user accounts, and to maintain key values that can be used to restore the state of the data manager should a failure occur. The lock managers are used to control concurrent access.

To increase the overall dependability of the system, the constituent FS atomic objects would typically be constructed using fault-tolerance techniques to increase their failure resilience. For example, the transaction and data managers might use recovery protocols to ensure that data in the system is restored to a consistent state following failure. Similarly, stable storage might be replicated to increase its failure resilience. The failure notification aspect of FS atomic objects is used to allow objects to react to the

and describes how its features facilitate the implementation of fail-stop atomic objects using commonly accepted techniques. The use of the language is illustrated in Section 4 with the presentation of a data manager and associated stable storage. Section 5 provides an overview of the implementation strategy currently being used to construct the language, while Section 6 discusses the features that distinguish it from other languages proposed for fault-tolerant programming. Section 7 offers some conclusions.

2 Fail-Stop Atomic Objects

As mentioned, our programming model is based on the abstraction of a fail-stop (or FS) atomic object. Such an object contains one or more threads of execution, which implement a collection of operations that are exported and made available for invocation by other FS atomic objects. Higher-level FS atomic objects may, in turn, be constructed by composition. When invoked, an operation exported by an FS atomic object executes as an atomic action as long as no failures occur. In the event of a failure that results in the loss of the object's ability to implement operation atomicity, a detectable *failure notification* is generated; the status of an operation being executed when such a failure notification occurs is indeterminate. This notification could be generated, for example, if the processor on which the object is executing crashes, or if a lower-level constituent object fails. This notification aspect, which is one of the properties that distinguishes FS atomic objects from regular atomic objects [Lis85], is intended to allow for the possibility that the abstraction might, in fact, fail to hold under certain circumstances. This reflects the reality that there is always some non-zero probability that an abstraction may not be maintained should, for example, multiple failures occur simultaneously. Hence, the analogy to fail-stop processors implied by the term "fail-stop atomic objects" is strong: in both cases, either the abstraction is maintained (atomic object or processors) or notification is provided.

A fault-tolerant, distributed system can be realized by a collection of FS atomic objects organized along the lines of functional dependencies. For example, an FS atomic object implementing the services of a transaction manager may use the operations exported by another FS atomic object implementing the abstraction of stable storage [Lam81]. These dependencies can be defined more formally using the *depends* relation given in [Cri91]. In particular, an FS atomic object u is said to *depend* on another object v if the correctness of u's behavior depends on the correctness of v's behavior. Thus, the failure of v may result in the failure of u, which in turn can lead to the failure of other objects that depend on u. Increasing the dependability of a distributed system organized in this way is done by decreasing the probability of failure of the FS atomic objects out of which it is composed. We informally call this measure the *failure resilience* of an object.

The failure resilience of an FS atomic object can be increased using standard fault-tolerance techniques based on the exploitation of redundancy. For example, an object can be replicated to create a new FS atomic object with greater failure resilience. This replication can either be active, where the states of all replicas remain consistent, or passive, where one replica is a primary and others remain quiescent until a failure occurs. Or, an FS atomic object can contain a recovery protocol that would be executed upon restart following a failure to complete the state transformation that was in effect when the failure occurred. The applicability of each of these techniques depends on the details of the system or the application being implemented.

As an example of how a typical fault-tolerant system might be structured using FS atomic objects, consider the simple distributed banking system shown in Figure 1. Each box represents an FS atomic object, with the dependencies between objects represented by arrows. User accounts are assumed to be partitioned across two processors, with each data manager object managing the collection of accounts on

1 Introduction

Ensuring the *dependability* of computer systems—that is, that the system delivers services on which people can rely [Lap91]—is an increasingly important issue. One key aspect of this problem is the development of techniques and support systems for constructing *fault-tolerant, distributed programs* that can continue to execute despite the failure of one or more processors in a distributed system. Such programs are intimately tied to the problem of increasing dependability, since many systems in a variety of areas ranging from databases to process control are, in fact, fault-tolerant distributed programs of one kind or another.

Constructing programs of this type is undeniably difficult, which has led to research in a variety of areas aimed at systematizing and simplifying various aspects of the task. For instance, *failure models* have been developed as a means for precisely specifying assumptions made about the possible effect of failures. Examples of popular failure models include fail-stop [SS83], timing [Cri91], fail-silent [PVB+88], and Byzantine [LSP82]. Another area has been the development of *programming paradigms*, which simplify the development of certain types of fault-tolerant, distributed programs by providing canonical system organization techniques and abstractions for that type of problem. Examples of popular programming paradigms include the object/action model [Gra86], the restartable action paradigm [Lam81], and the replicated state machine approach [Sch90].

In this paper, we focus on a third area related to simplifying the construction of fault-tolerant distributed programs, that of providing adequate programming language support. Specifically, we present the design of FT-SR, a programming language based on SR [AOC+88] that is designed for writing fault-tolerant, distributed systems. FT-SR is unique in that it has been designed to be a *multi-paradigm language*, that is, a language that can support equally well *any* of the multiple programming paradigms that have been developed for this type of system. This is done by providing support for constructing *fail-stop atomic objects*. Like an atomic object, execution of operations implemented by these objects is atomic, i.e., all or nothing. However, unlike standard definitions of atomic objects, we allow the possibility that the atomicity abstraction may fail, for example, should the redundancy being used in its implementation be exhausted. In this case, the object stops executing in a detectable way. This new type of object is a common link between paradigms since each of their fundamental abstractions is actually a fail-stop atomic object, with apparent differences being due to either varying failure assumptions or implementation techniques. Here, we concentrate on processors with fail-silent semantics, although the approach generalizes to other failure models as well.

The orientation towards a multi-paradigm approach distinguishes FT-SR from other languages [Lis85, EFH82, LW85], language extensions [SCP91, CGR86, KU87] and language libraries [BSS91, Coo85, PS88, HW87] related to fault-tolerance, which are typically oriented around a particular paradigm. Support for a single paradigm has been shown to be constraining in many situations [Bal91], and is particularly inappropriate for constructing systems, where different paradigms may be used at different levels of abstraction. Moreover, the development of such a multi-paradigm language for fault-tolerant programming can be viewed as analogous to the evolution of standard distributed programming languages, which have progressed from languages such as CSP [Hoa78] and Concurrent Pascal [BH75] that support only a single synchronization paradigm to those such as Ada [DoD83] and SR that support multiple approaches.

This paper is organized as follows. In Section 2, we first describe fail-stop atomic objects and the programming model that results from considering these as the primary abstraction. We also argue that this model is a "lowest common denominator" for the various programming paradigms, and hence, a realistic basis for the design of a multi-paradigm language. Section 3 then outlines the design of FT-SR

A Multi-Paradigm Programming Language for Constructing Fault-Tolerant, Distributed Systems[1]

Richard D. Schlichting Vicraj T. Thomas

TR 91-24a

Abstract

The design of FT-SR, a programming language oriented towards constructing fault-tolerant distributed systems, is presented. The language, which is based on the existing SR language, is unique in that it has been designed to be a multi-paradigm language that can support equally well any of the various programming paradigms that have been developed for this type of system. These paradigms include the object/action model, the restartable action paradigm, and the replicated state machine approach. To do this, the language is designed to support the implementation of systems modeled as collections of fail-stop atomic objects; such objects either execute operations as atomic actions, or fail in a detectable way. It is argued that this model forms a common link among the various paradigms and hence, is a realistic basis for a multi-paradigm language. An example program consisting of a data manager and its associated stable storage is also given; the manager is built using the restartable action paradigm, while the stable storage is structured using the replicated state machine approach. Finally, the implementation strategy for the language runtime system is discussed.

March 17, 1992

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This work supported in part by the National Science Foundation under grants CCR-8811923 and CCR-9003161, and the Office of Naval Research under grant N00014-91-J-1015.

A Multi-Paradigm Programming Language for Constructing Fault-Tolerant, Distributed Systems

Richard D. Schlichting Vicraj T. Thomas

TR 91-24a