Introduction

KENNETH P. BIRMAN AND ROBBERT VAN RENESSE

It is far easier to develop a computer program that performs adequately than to develop one that performs *well*. Nowhere is this more evident than in distributed computing systems (i.e., software for networks), where a single application may need to be split into many small pieces. This type of computing system may have such a huge number of components that the "system" takes on an identity that is almost independent of the components of which it is assembled.

In a distributed system, knowing that all the components are behaving correctly is not enough to guarantee that the system is behaving correctly. The "system" must ensure that related actions taken by different components are mutually consistent, and that actions are timely and correct even when multiple components cooperate to perform the actions, and hence must address concerns that span multiple components. Complicating the issue, distributed systems may need to continue running even if some of their components crash, malfunction in other ways, or even attempt to compromise the system in an aggressive way–even if this occurs at a highly inopportune time, such as during an action that has locked out other computing actions. Distributed systems must be able to manage themselves automatically, without the help of a human operator to monitor the status of constituent application programs, to restart critical components after failure, to balance loads, or to shut down gracefully when a node must be halted for maintenance.

Our interest lies in this class of tightly integrated, reliable, highly resilient distributed computing systems. How can they be developed? Why should we believe that they will operate reliably, and what assumptions do we make about the environment when making them reliable? What factors limit performance? How well do the necessary technologies scale? What are the costs of reliability?

The reliability–or lack thereof–of modern computer networks and distributed systems will determine the degree of acceptance such systems ultimately gain. Many corporations are investing enormous sums in their computing infrastructure, building larger and larger networks that are critical to their livelihood. The Federal Government is echoing this with massive investment in *data highways*, a national high-speed computer-to-computer communication infrastructure. Business reports carry stories about *corporate memory*–information artifacts resulting from the use of computers in product development, marketing, and support–and speculate about the development of new kinds of corporate assets. Indeed, communications and computing are often touted as a sort of magic bullet for the economy, an engine

that will drive vast economic growth and from which new information-oriented businesses, products, and services will emerge.

Only if distributed computing systems can be made extremely reliable can this ambitious vision be realized. With the opportunities of high performance communication and computing (HPCC) comes the obligation to engineer distributed software to exacting standards of reliability.

The fundamental problem is that modern distributed computing technologies have not responded to this challenge. In the mid 1990s, as we write this introduction, most distributed computing systems are developed using techniques little different than those used in the earliest computer network software of the late 1970s. Application software is almost entirely decoupled, and little help exists for the developer faced with complex distributed correctness goals. Distributed systems are fragile, requiring substantial human resources simply to keep them running, and are easily prodded into system wide failure or inconsistent behavior by configuration changes or individual component outages. Unless there is a major advance in the way that distributed systems are engineered, the opportunity of HPCC will go unrealized.

The Isis Toolkit is a response to this challenge. In 1983, when we started the project, we were struck by the difficulty of developing software for dynamic, failure-prone, high performance settings. The toolkit encapsulates the difficult aspects of distributed computing in an easily used programming environment that is compatible with the standards and yet addresses the special reliability issues encountered in distributed settings. As the papers collected for this textbook will show, Isis has enjoyed considerable success. Of course, no system can be a complete success, but we think that the style of distributed computing that Isis embodies is now well established and here to stay. The potential for the future staggers the imagination.

1.1 Categorizing computing systems

The properties and functions of "systems" software have traditionally been driven by the needs of applications programs. These, in turn, are generally determined and limited by the characteristics and cost of hardware components. This perspective makes it possible to organize the evolution of systems software using a number of broad categories, according to increasing scale, growing numbers of software components, and the growing emphasis on the sum of the parts–the whole–rather than on the individual constituents. Here is one example of such a categorization:

- *Time-shared mainframe computer systems.* Early multiuser computer systems were primarily shared mainframes, which offered a way to share resources among a collection of users and application programs. The "classical" focus of operating systems research has been on the techniques needed to build correctly operating systems for such settings (hence, on synchronization and concurrency, mutual exclusion, scheduling, virtual memory, memory hierarchy, etc.), and on the best ways to present such systems to the programmer (process abstraction, virtual memory, file systems, communication streams and pipes, capabilities, and so forth). The period of greatest activity for this type of work was from 1965 to 1980.

- *Networked time-shared mainframes.* Early networks linked mainframes and time-shared minicomputers, focusing on straightforward network applications: file transfer, remote login, electronic mail and bulletin board services. A body of distributed systems research emerged from this period, concerned with network management, packet routing, and the most appropriate layer emerged during this period in which to do flow control and error correction, along with software such as telnet, ftp, uucp, net email, and netnews. The development of these systems peaked during the period 1980–85.

- *Client/server workstation architecture.* In the mid 1980s, it became possible to put the power of a mainframe on an individual's desktop. The term client/server computing emerged to describe the resulting network architecture, which can be visualized as rings of clients surrounding servers that manage files, database, and other resources, interconnected by wide area network links.

  Although client/server systems drew heavily on existing technology for distributed computing and time-sharing, ideas such as window-based user interfaces, highly responsive interactive applications, and desktop publishing also emerged during this period. One consequence was that increasing pressure was placed on the programming methodologies used to develop distributed software, because developers recognized that the effort being invested in the distributed side of their applications was out of proportion to the effort required to develop other aspects. Unfortunately, although the rate of advancement was rapid for nondistributed software, this was not matched by corresponding advances in distributed systems. For example, one can point to powerful toolkits for exploiting high resolution displays and interactive window systems, elaborate desktop publishing and engineering environments, and powerful visualization aids like spreadsheets. Toolkits for developing reliable distributed software, however, proved surprisingly difficult to develop, and lagged far behind.

  The most successful style of network computing was also the simplest: in which the applications are decoupled from one another, and interact indirectly through files in the file server, or by exchanging electronic mail or postings through subject-oriented bulletin boards, but in which direct program-to-program communication was minimized. Substantial program development libraries emerged in support of this style of distributed computing, including important, widely popular standards like the ONC[1] and DCE[2] environments. Distributed object-orientation, partly a response to the slow progress and complexity of distributed computing, yielded standards like CORBA, the *common object request broker architecture* of the OMG standards body. But this, too, employs a very simple, loosely coupled computing model. The user facing a problem that combines reliability considerations with a requirement for tight coupling between system components would find little help in any of these programming styles or environments.

- *Tightly coupled distributed computing systems.* Some types of distributed computing systems require direct cooperation between the programs of which they are composed.

[1] Open Network Computing, developed by Sun Microsystems.
[2] Distributed Computing Environment, developed by OSF.

In these systems, a network of distributed computing platforms provides tightly integrated services, creating the illusion that the system is in fact not distributed. These systems need ways to dynamically reconfigure themselves to remain available despite failures and recoveries, and to balance loads as demands on the system change. Good examples are the software environments built in support of trading and brokerage applications, the systems used to control large telecommunications applications involving large numbers of switching points, and next-generation air-traffic control systems.

When Isis[3] emerged in the late 1980s, the target was the development of this class of distributed application. Isis was conceived after a group of us at Cornell University developed a complex, tightly coupled distributed computing system between 1983 and 1985. This early system, which supported "resilient objects," was noteworthy primarily because of the way it had been engineered. It occurred to us to try and move beyond this initial foray into distributed computing by breaking out the enabling layer of software in the form of a general purpose toolkit that could be used in developing a wide variety of complex distributed systems. Encapsulating the difficult synchronization and fault tolerance problems in an easily used toolkit, enabled a style of distributed application development that normally would require such substantial resources as to be feasible only for distributed systems software engineers. Moreover, the toolkit offered a rigorously specified, easily understood programming model called *virtual synchrony*, and this turned out to be a key to solving a wide range of highly varied distributed computing problems.

During the late 1980s, a rapidly growing user community emerged for the initial academic version of the toolkit, ultimately spawning a commercialized implementation that has been widely adopted by industry. Today, Isis is used in hundreds of settings ranging from financial trading to VLSI factory floor automation and advanced telecommunications. Applications of the system include some of the most ambitious distributed problems ever attempted, and the growth rate of the market exceeds our most optimistic early estimates.

A rarity until the late 1980s, tightly coupled distributed computing systems are now entering a phase of rapid expansion as large numbers of organizations and enterprises exploit general purpose computing platforms in settings that had previously been treated using handcrafted, one-time solutions.

- *Enterprise computing.* Corporations and other large organizations face enormous pressure to streamline their operations, make better use of information, and respond more rapidly to changing opportunities and circumstances. This leads to a new vision of communication and information melded into a seamless whole, of new ways of do-

[3]Amr El Abbadi, now a faculty member at U.C. Santa Barbara, contributed the name of the project. Isis was an Egyptian goddess, consort of Osiris (who later came to rule the underworld to which the living passed after entombment). In an epic battle, Osiris fought Seth, an evil god, and Osiris was defeated. His body, torn into shreds, was scattered over the Nile delta. Isis gathered the pieces and mummified Osiris, anointing him with sacred oils and restoring him to life–an apt image, Amr suggested, for a fault-tolerance technology! Later, Isis and Osiris had a son, Horus, who defeated Seth; this led us to pick the name Horus for our work on a next-generation of the Isis Toolkit.

ing business and carrying out social interactions in which computing and multimedia technologies knit worldwide organizations more and more tightly together. The term *enterprise computing* is used to refer to such an approach. An enterprise computing system is a single, highly integrated distributed system spanning potentially all the nodes in an international corporation or *enterprise*–tens or hundreds of thousands of computing platforms of widely varied capabilities, local and wide area networks of highly varying delay and speed characteristics, and applications ranging from conventional desktop publishing to demanding distributed processing and control.

The development of successful enterprise computing systems will be one of the major research and commercial battlegrounds of the late 1990s. Applications such as remote collaboration using real-time video data display systems, three-dimensional image interpretation, wide area information retrieval, and so forth, will drive the advances in this field. Technology developments such as the rapidly approaching gigaflop computing platforms, massively parallel processing, and ATM gigabit networking will have the power to support such applications. The challenge to systems builders concerns harnessing and presenting this computing power in a way that application developers can exploit effectively.

- *Ubiquitous computing.* Enterprise computing will push contemporary desktop and distributed computing to its limits. Beyond this limit lies a new kind of computing, exploiting powerful computers packaged within liveboards, hand-held devices, vehicles, appliances, televisions, and even mundane objects like light switches. Ubiquitous computing systems share properties of enterprise computing, but with a far higher concentration of devices, used for very different types of applications, and in a world of higher speed and rather specialized demands. As we write this, ubiquitous computing still seems far in the future. Realistically, one might expect research in this area to gain speed late in the 1990s, with the first major successes early in the 2000s. The development of distributed software environments for programming such systems stands forth as one of the great opportunities and technological challenges of the 1990s.

1.2 A philosophical evolution

It is striking to reflect upon the extent to which the philosophy of distributed computing has evolved over the past twenty years. Machines have become smaller and yet more powerful. Application software has become more powerful, but has diminished in importance when one considers the enormous numbers of application programs that must join forces to provide enterprise computing solutions, or to control a factory production line or a telecommunications switching system. The trend is clear: future systems will include staggering numbers of individual computing systems which must somehow cooperate to provide integrated, consistent functions. Future systems must achieve high levels of reliability and security, and must dynamically adapt to changing environmental conditions, workloads, and hardware. A new style of computing is being born that is concerned not just with what individual computers do, but with how they cooperate and how they behave in the aggregate.

The networking technologies of the past were "self centered," concerned primarily with individual programs, individual servers, and with concealing the distributed nature of the environment so that the applications developer could work as if on a dedicated mainframe. Technologies like *streams*, DCE, ONC, and the Open Systems Interconnection (OSI) architecture elevated this approach to standards that are supported by most vendors on most machines. Yet, by failing to confront the increasingly integrated nature of the applications being built, such approaches ultimately fail the developer: they push so much of the difficult aspect of distributed computing into the hands of the developers, that few developers can deal with the complexities of large scale distributed environments. Moreover, as systems scale to become truly large, the need for standards enters: it is not enough to build a reliable application; one also needs to run that application in a reliable infrastructure and to be able to trust the reliability of other applications developed by other users.

The demand and the challenge, then, are to devise a new concept of distributed computing offering tools for use throughout large-scale networks that simplify the task of the applications programmer and impose a high degree of uniformity. With this approach, the developer can contribute reliable technologies to a reliable technology base, adding new components to a preexisting, tightly integrated whole. Given distributed computing tools that address fault tolerance, security, and other crucial concerns in an easily understood manner, we can finally begin to look beyond the desktop to the concerns and issues of deploying enterprise computing systems. Distributed computing is far more than an issue of communication and transparency: it is an entire philosophy of how complex systems should be developed from large numbers of components. It will require an evolutionary change in the way that we approach computing systems.

1.3 Goals of this book

Our goal in this book is to place the Isis system into perspective with regard to the sweeping technology advances and varied application domains in which a need for tightly coupled reliable computing arises. The papers selected range from very technical results documenting various internal aspects of Isis and Horus, to much less technical papers concerned with how we and others have used these systems to develop closely coupled and fault tolerant application environments. We have tried to strike a balance between theory and practice.

Isis originated from research on *tools* for building distributed computing and process control applications, and as we will see, basically combined ideas the theory of distributed computing (atomic broadcast and logical time), the state of the art in desktop distributed systems (process groups), and from other areas (database transactions and serializability). The resulting software development environment turns out to be a particularly effective base for building fault tolerant distributed services and for ensuring that the behavior of a distributed system will be consistent even as it reconfigures itself to react to failures and other types of events.

Horus goes beyond Isis by offering a lightweight, more flexible presentation of the latter's ideas in a form that integrates well with new microkernel technologies and is suitable for the distributed control and enterprise computing applications of the near future. Horus implements many Isis ideas more efficiently, while drawing on concepts originating in modern

operating systems, and on object-oriented modularity ideas. Horus also addresses security–a topic overlooked in our early work on Isis, and will include a real-time computing subsystem, aspects of which are under development by Carlos Almeida, Keith Marzullo, and Paulo Veríssimo (see Paper 10).

The remainder of this introduction is organized as follows: begin by discussing distributed computing in historical terms, in Section 1.4. Building on this, Section 1.5 discusses the evolution of our work in this area, summarizing the contributions of the major papers on Isis and placing them in the context of other work that influenced ours. As Isis began to mature, we used it to develop a number of applications. These are described in Section 1.6, papers on many of which appear later in the book. We also began to elaborate the theory underlying Isis, as discussed in Section 1.7. Work on the Horus system is, as discussed in Sections 1.8 and 1.9. At the time of this writing, Horus is in an early stage of completion and has yet to be applied to serious problems, but we are able to report on our initial experiences with the new system.

1.4 A brief history of distributed computing

1.4.1 Network computing systems

It may be useful to contrast *distributed computing*, as we use the term throughout our work on Isis, with *networking* or *telecommunications*. Although these terms are sometimes employed interchangeably, it has become common to say that a networked application is one consisting of comparatively independent programs communicating over some form of message-oriented substrate. For example, a user who logs onto a machine remotely could be said to be running networked applications, merely by virtue of using a remote connection to communicate with the machine on which the applications are running.

The concerns of a network computing system are normally described in terms of the OSI layering. In this descriptive structure, successive layers of software abstract the details of physical data transmission, packet transmission (small fixed-size messages), message transmission, routing, data representation, program addressing, and session management. Each layer is built over the services of the underlying layer, with the goal of supporting applications such as a file transfer protocol, a remote login protocol, and electronic mail, as well as other application-specific protocols. Examples of network architectures matching this model are the OSI network, and the TCP/IP protocol suite, which actually does not comply precisely with the OSI standards but is nonetheless closely matched with the OSI architecture.

Two aspects of the OSI philosophy are particularly relevant to us here. First, the technology is very *point-to-point* in its view of communication. That is, the goal, in some sense, is to allow a program, operating at location A, to pass data or send requests to a program at location B. Issues spanning more than a pair of programs are simply not considered. Secondly, the technology seeks to hide the network from the application through what are often called *transparencies*. The idea is that if the user could interact with a remote server or resource as if it was local–as if the network was not present, or was *transparent*–application development would be greatly simplified. And indeed it is, provided that the

application matches the sorts of point-to-point applications for which the OSI structure was conceived.

Transparency, in this setting, refers to a technology that conceals details of the underlying environment. A network-transparent technology clearly has beneficial attributes, since it simplifies the programmer's task–to the extent that those underlying details aren't the issue. But transparency can also hurt: a very transparent system may also be one in which it is hard to take advantage of important properties of the environment. Indeed, in hiding the presense of the network, a very transparent distributed computing technology also makes it hard to exploit distribution for parallel processing, fault-tolerance, and other purposes.

1.4.2 Remote procedure calls

One of the most important of the transparency technologies is *remote call.* Often referred to as RPC [BN84], this is an approach in which *client* programs issue procedure calls to procedures defined by remote *server* programs. The complexity of making connections and marshalling data in and out of messages is hidden by collections of *stubs* which mimic the interface of the remote procedure they will invoke, except, however, for using different error semantics.

RPC has been popularized by two major distributed computing environments: ONC, the RPC technology that was developed by SUN Microsystems to support the Network File System (NFS) standard, and DCE, the ambitious RPC and distributed computing environment developed by the Open Software Foundation.

Using RPC, network operating systems have developed a small set of distributed services that became standard, complementing the network file system services, and providing a network applications development environment. These services normally include a synchronized clock service (one that keeps the clocks on workstations synchronized to some reasonable precision), a name service (used to find the network addresses of servers), and a security service (used to establish secure communication channels and to obtain trusted information about the identity of participants).

Network operating systems have been hugely successful. Their introduction created new markets in which hardware and software could be sold, and they completely revolutionized day-to-day life in modern corporations, universities, and government. Immense numbers of applications for these kinds of systems emerged during the early 1980s, and they rapidly became a primary area of expansion for the computing hardware market–a development that caused many older hardware vendors to falter,because they were unable to shift to the sort of corporate mentality demanded by the lightweight, flexible, inexpensive computing environments. Meanwhile, advances in computing hardware caused workstation performance to approach and finally eclipse that of traditional mainframe computers. Except in certain specialized settings (such as massively parallel supercomputing), an era of networked computing was firmly established by the late 1980s, and continues into the present.

1.4.3 Distributed computing systems

In this book, we will use the term *distributed computing* to refer to a type of computing in which programs are tightly coupled: they coordinate their actions, know about the membership of the overall system, and may even have knowledge of one another's states. This is in sharp contrast with *networked programs*, where programs access resources over a network, perhaps from a file server, but do not explicitly cooperate or otherwise know about one another. To understand the issue, consider the following two examples.

- Programs A and B interact with file server S, sharing data through a file that A writes and that B reads and modifies. The programs were written as two components of a single application, and are designed to be run in order: first A, then B. Normally, A and B would not run on the same machine.

- Programs A and B cooperate as the *primary* and *backup* for a high reliability financial analysis program, providing the broker with critical advice needed to carry out a sophisticated trading strategy. As the primary, A monitors various data feeds and performs a complex computation, periodically checkpointing its state to B. B monitors A. If A fails, B takes over by starting execution at A's last checkpoint and resuming the same computation; the takeover might be noticeable but the disruption is minimized by careful application design. The application program C displays data to the broker and issues requests to the primary A, switching to B if A is observed to fail.

At a glance, it may appear that both of these examples describe typical networked computing applications, and that neither is "more distributed" or otherwise distinguished in any deep sense from the other. However, on closer inspection, one sees that this is not necessarily so. The first type of system is distributed by virtue of its design and intended mode of operation, but the programs composing it are completely independent and interact only indirectly, through the data files they manipulate. Structures such as this are common in the type of networked computing that has been most prevalent during the decade of the 1980s; so much so that they have even been given a name: *stateless* distributed systems. This does not mean is that the programs embody no state, but rather that their states are independent–they share no *common* state. Program A doesn't know the machine on which program B will run, for example, or at least doesn't need to know this to operate correctly; similarly, program B has no explicit knowledge of where and when program A was executed.

Network computing systems turn out to be ideal environments for stateless distributed computing. The sorts of tools they offer are well matched to the patterns of interaction supported in modern network programming environments. Such environments are often lauded for being transparent in the sense that the user is practically unaware that the network is present at all.

The second example illustrates a much more explicit style of cooperation and coordination between the programs involved–it is what we could call it a tightly coupled distributed application. Here, programs A, B, and C need to be explicitly "aware" of one another, and to monitor one another's state. Moreover, a form of distributed state is present, namely, the respective views that the programs have of membership in the system, and the correctness of the application depends upon this information being consistent. Suppose, for example,

that program C incorrectly decides that program A has failed–and that program B does not reach this decision. This could easily happen in a network, since a transient problem could prevent A from communicating with C for just long enough to cause C to conclude that Ahas failed, without disrupting communication from A to B long enough to trigger the same decision in B. Now A, B, and C are in inconsistent states: C believes that B is the primary and will provide analysis for the broker, but B believes that it is still the backup, and A believes that it is still the primary. The broker is thus denied service because of an inconsistency.

This example illustrates many of the issues that motivated the Isis project. Isis is concerned with consistency in tightly coupled distributed systems–with guaranteeing that collections of programs running in a networked environment can cooperate to share data, subdivide a task, dynamically adapt to failures, and in general contrive to provide coordinated behavior despite having multiple cooperating components. Distributed computing of this sort is not necessary in some settings. However, this need arises in a growing number of networked applications, and the basic idea behind Isis is to provide the developer with tools to solve these kinds of problems in a way that will result in highly reliable, high performance applications software.

To summarize, one can crudely classify distributed computing systems into two groups. The applications in one category are loosely coupled, interacting with servers using remote procedure calls but not with one another. The applications in the second category are tightly coupled, explicitly cooperating to perform a task that none could carry out in isolation.

Modern distributed computer systems confirm this categorization. When an individual sends electronic mail, a series of point-to-point interactions occur as the email message is forwarded from the editing program in which it was composed to the mailbox file of the ultimate recipient. As the mail advances through each step on the way to its destination, program-to-program communication occurs, but the programs involved soon forget about their interactions: once the mail has been delivered, no memory of the interaction remains at all. This is typical of a loosely coupled, *stateless* communication pattern. Such interactions are typical of the approach favored in distributed computing environments such as DCE, ONC, or even the new object-oriented environments CORBA and OMG.

On the other hand, consider a financial application in which a small group of foreign currency traders cooperate to manage a billion-dollar investment portfolio for a bank. On the screens presented to each trader are a number of trading parameters and recommendations, all consistent and coordinated across the group. If the members of the group are split between major financial centers (Zurich, London, New York, and Tokyo), support for this style of computing will require a distributed application with tightly coupled cooperating components at multiple locations that synchronize their actions and communicate important data in such a way as to maintain the consistency of the recommendations and reporting to traders, while also tolerating failures and dynamically adjusting to balance load.

Here, one sees a need for group communication and computing: a group of programs that need to coordinate their actions and replicate various forms of financial data for display to the local trader. Cooperative algorithms are often important in such applications. For example, suppose that the bank wishes to limit its risks by never placing more than 20% of its cash in any single currency. If Yen pricing suddenly becomes appealing, how will

traders coordinate their purchases so as to respect this constraint? To the degree that the traders rely on the computer system to advise them correctly–allowing trading up to the limit, but never exceeding it–they will insist on reliable, consistent behavior. Anything less could impose unacceptable risks on the bank.

The same sorts of problems are seen in many distributed computing systems, such as for air-traffic control, for medical monitoring in hospitals, in support of telecommunications routing, and for huge numbers of mission-critical, inherently distributed applications.

Problems characterized by this type of close cooperation and tight coupling are not particularly simplified by tools such as the ones in DCE or ONC; nor do object-oriented computing technologies like CORBA and OMG help very much. All of these standards are focused on how one program should interact with another – how programs should be named in distributed systems, how to find those names, how to package requests and send them, and how to unpack the responses. These technologies strongly encourage the decomposition of a network application into large numbers of components. Yet nowhere in these standards does one find a technology addressing the need to synchronize and coordinate the actions of those components, to organize components into groups, and to ensure that the overall system will behave in a consistent, reliable manner. Isis addresses precisely these problems, and would normally be used as an adjunct to a technology like DCE or CORBA. For example, Isis can be used to build a fault tolerant server whose behavior, as seen by clients, would be indistinguishable from that of a nondistributed but highly reliable server. DCE and CORBA explain how one would issue requests to such a server; Isis explains how to make the server reliable.

1.4.4 Controversy

Although it may not be evident from the papers collected in this book, Isis has been embroiled in some degree of controversy almost since we proposed the approach. To understand the reasons for this, it may be helpful to describe a very different controversy, between database systems developers and operating systems developers, which emerged in the 1970s and persists to the present.

Early in the development of computer systems, no real distinction was drawn between databases and operating systems. They were all systems of one kind or another, and database systems were often considered to be a particularly fancy class of applications that made intensive specialized use of files, but were still very much a part of the *systems* community.

A series of events changed this state of friendly cohabitation. First, the database community and operating systems community differed on how to deal with concurrency and fault tolerance. Database researchers proposed that concurrency be addressed through a model called *transactional serializability*, in which database applications were described as transactions that terminate by committing or aborting atomically. The basic goal of a database file server was defined to be the rapid execution of transactions, by interleaving the servicing of operations from independent transactions (to increase concurrency) in ways that preserve the abstraction of independent execution, through serializability. A substantial theory evolved for database concurrency control and transaction management, along with an elaborate theory of transactional query decomposition and data structure and memory

management. All of this began to place strong demands on the underlying operating systems, and it became common to hear calls for database operating systems and other special-purpose solutions.

The operating systems community, meanwhile, followed a different path. By defining the transactional applications to be *database applications*, this community became focused on the class of applications that are *not* easily characterized in transactional terms. These lack any clear pattern of data access, and any clear notion of a commit point. Although they may be multithreaded, operating systems applications generally need mutual exclusion mechanisms rather than full-fledged database concurrency control schemes, and patterns of file and memory access turn out to be very dissimilar from what the database community considered typical. These trends led the operating systems community to pursue topics such as high speed communication, virtual and shared memory abstractions, lightweight thread support, and RPC, but without the need to maintain any sort of overall system execution model, or to prove properties of concurrent software beyond properties such as noninterference and liveness.

This trend was particularly acute in the case of networked computing and distributed computing, somewhat paradoxically because of the substantial body of research on the theory of distributed computing that was being developed at the time. Curiously, this theory yielded almost entirely negative results concerning *real-world* distributed systems, namely, those in which communication speeds and reaction times could not be predicted and might be unbounded. For asynchronous systems, most of what was known was that certain problems were not solvable, at least not in full generality. Beyond this, there was work on problems such as deadlock detection and reliable communication, but not much of it was ever implemented or shown to be useful.

On the other hand, a substantial amount of work was devoted by the theory community to understanding a class of idealized distributed systems, described by the *synchronous* model. In this model, which no real-world system can achieve, processors operate in rounds, exchanging messages with one another within each round, and suffering a bewildering variety of possible failure types (omission to send messages, omission to receive messages, timing failures, crash failures, and arbitrary, potentially malicious, Byzantine failures). While much progress was made on understanding fundamental limitations and algorithmic approaches through this intentionally *unrealistic* model, not much of what was done turned out to be applicable to *real* systems.

Thus, not very surprisingly, the operating systems community turned away from theory and models, and away from abstractions, and focused on pragmatic issues: building network software, understanding network performance issues, and building better and faster network operating systems.

As we will see below, Isis entered this picture by proposing an extension to a type of *process group* computing first introduced in Stanford's V project, by Cheriton and Zwaenepoel [CZ85]. These researchers suggested that by explicitly supporting a group programming model in the operating system, certain classes of applications could be simplified. They gave a number of examples, including game playing applications, replicated servers within the operating system, and publish/subscribe communications (whereby programs communicate by publishing messages using a *subject*, to which interested recipients *subscribe*).

Isis took process groups one step further, by adding a distributed execution model that draws on ideas from database serializability and atomicity, although without introducing a notion of transactions. In Isis, the programmer knows how process groups will behave, even in the presense of failures and concurrency, and is offered a collection of flexible programming tools with which tasks such as primary/backup computation and replicated data management can be solved. This model, called *virtual synchrony*, pervades the entire Isis environment, offering the process group programmer a simplified, idealized view of the network, and insulating him or her from the complexity of solving problems like agreement on system membership. In Isis, these hard problems are handled by the system.

Further complicating the issue, the theory underlying systems like Isis has emerged rather slowly. The theoretical basis of this class of systems turns out to be closely related to what was understood from prior work, but is just different enough so that prior work can't be directly applied to Isis. Thus, Isis could be said to have introduced a sort of halfway rigor on top of the earlier V work; rigor that improved with time (fortunately, no major holes in the approach have surfaced!), but that certainly didn't make the system easy to understand on first exposure.

All of this lead to a certain degree of controversy, which continues today. The first reaction of the operating systems community was that Isis didn't seem terribly relevant to the problems encountered in networking. There was a reaction against the introduction of a model (by the hardcore pragmatists), and a reaction of confusion about the model itself (by the hardcore theory community). Yet, the system proved to be an effective tool for developing reliable distributed software, and gradually amassed a substantial user base. With time, the performance of the system has improved to the degree that it now competes well with any alternative, the theory has been filled in to the degree that the theory community has begun to accept the work; and the flexibility and presentation of the system have improved to the degree that it can coexist more easily with conventional communications technologies, such as RPC.

As we write this introduction, the approach has certainly found widening acceptance, but many aspects remain disputed. Certainly, the degree of *use* of the system has grown dramatically, and has come to include some extremely demanding applications. Yet it would be hard to claim that Isis has really entered the mainstream. Perhaps, like database systems, distributed computing systems will eventually branch away from other forms of networking and begin to be viewed as an independent area buttressed by their own body of research.

1.5 Isis background and development

Whereas the previous section focused on historical background, this section discusses distributed computing, and process group computing, in more technical terms. Our goal is to first delineate the major technical issues that need to be addressed in these systems, then to point both to our own work (included later in this book) and to other relevant work, which has had a substantial impact on how we now view the system and how it has evolved over time.

Process group computing environments normally offer a set of basic services that includes the following:

- A way to create, *join*, and leave groups.
- A name-space mapping from symbolic names to some form of communication handle on the group bound to that name.
- A way to send messages to a group, and to collect replies from some (or all) of its members if desired (often called *group multicast*, or *group broadcast*"–we will use the former term).
- A way to determine the current membership of a group.
- Group programming tools, which use groups to solve practical problems.

Group computing systems differ in the degree of guarantees that they provide for these operations, in the levels of reliability they achieve, and in the way that the basic mechanisms are implemented.

The earliest system to use process groups was Stanford's V system [CZ85]. In the V approach, groups are used for applications such as publication of data and interaction with distributed services and games. Communication is reliable, but in uncommon situations, such as when multiple processes issue nearly simultaneous group multicasts, the exact sequence of events seen by different members of a single group might not be identical. For example, if a group's membership is changing just as a message is sent, the V system might fail to send a message to the new member. Additionally, V will give up on transmission of a message after some number of retries have been attempted; thus, under some conditions, a message might not reach a process even though both sender and destination are operational. Different members may also see the same messages, but in different orders. The V type of group is intended for applications that expect communication to be reliable but would not actually behave incorrectly if unusual sequences of events caused slight violations of reliability.

Beyond the introduction of process group computing, the main focus of work within V was on scaling and performance of the basic multicast primitive. This resulted in the creation of a substantial body of technology, some of which has migrated into UNIX, such as the IP multicast mechanisms used by V to actually transmit packets to multiple destinations sharing a multiple-access medium [DC90].

The Isis system was strongly influenced by V, and in many ways is patterned after V. Where Isis differs is in its emphasis on fault tolerance and on providing very strong synchronization guarantees to the user, even when messages are sent to a group concurrently, or when events like membership changes and failures occur that can disrupt the normal flow of information. The idea in Isis was to provide the programmer with a much simplified model of how the network behaves, so that applications concerned with providing strong behavioral guarantees to the user could be developed in a simple, straightforward manner.

Of course, this requires Isis to be fault tolerant, which is achieved by replicating key information so that if a part of an application survives a failure, so do the Isis services on which it depends. Getting this to work was a big part of the Isis effort, and proving that it really does work forced us to do quite a bit of theoretical research in addition to building the system.

Subsequent to the development of its first version, the Isis system has evolved steadily over several years. The changes are reflected in the papers we have reprinted here, which include a description of protocols used in the first release of Isis as well as a more sophisticated set of protocols employed in a subsequent extension of the system. Although Isis was eventually reimplemented from scratch as the Horus system, the latter set of protocols remains at the core of the system.

In addition to V, Isis, and Horus, several other distributed systems have used group communication. These include Psync [Pet87], Delta-4 [Pow91], Advanced Automation System (developed by IBM for the US FAA), Lazy Replication [LLS90], Amoeba [KTFHB89], and Transis [ADKM92b]. These systems differ in the class of applications they are intended to support, in the properties of the group mechanisms they employ, and in the way that their group support is structured. Isis and Transis are the only systems to use virtual synchrony as their basic communication and execution model, but Lazy Replication has a well-defined execution model of its own, which differs from the Isis one in addressing persistent data explicitly.

All of this views Isis primarily as a system. It is also possible to regard the system primarily as a reliable multicast protocol suite, in which case the question arises of comparison with other multicast protocols. There is a vast literature relevant to this issue, which we discuss in more detail in Section 1.7, below.

1.6 Isis Applications

Since Isis was introduced in 1987, it has been used for a wide variety of distributed applications and research projects. We have had our direct experience with perhaps, 30 to 40 of these projects, but many of them are covered under some form of proprietary or restrictive disclosure, and hence are not suitable for discussion here. Others were never documented through the sorts of academic publications that one might cite or include in a collection like the present one. A result is that the applications section of this book is slanted towards work done at Cornell or by our immediate colleagues. While not representative of the broad picture, we believe that the papers we have selected convey a sense of the considerable range of problems to which this technology can be applied, and also a sense of the sorts of next-generation applications that are enabled by the availability of tools like Isis.

These applications include:

- Two network file systems: RNFS and Deceit [SBM89], which were built using reliable process groups to obtain fault-tolerant file access and load-balanced read-access behavior.

- A distributed system for reactive control and instrumentation, called Meta/Lomita (see Paper 17), which was developed as a tool for monitoring a distributed application or environment through software that reacts, automatically and in a fault tolerant manner, to events that require some form of system reconfiguration or reaction (hence the term *reactive control.)*

- Paralex, a system for automatically transforming a data-flow description of a parallel computation into a control structure that can be used for highly parallel, fault-tolerant execution of the application (see Paper 18).

- StormCast [Joh93], a distributed programming environment supporting a wide variety of environmental and weather monitoring application programs, which is being developed and deployed throughout Norway.

- The Isis Network Resource Manager, a batching/spooling system that dispatches jobs on matching machines and restarts jobs interrupted by failures (see Paper 16).

- The Isis Distributed News system, which supports a simple topic-based publish/subscribe paradigm for distributed communication in which subscribers and publishers are automatically linked by the system, with message ordering and fault-tolerance guarantees that simplify program development.

- The World Bank's IMIS system, which links a collection of databases into a single integrated financial planning data pool, supporting a sophisticated spreadsheet interface with parallel execution of financial queries (see Paper 19).

- The Data Acquisition program used by CERN in the LEP particle physics experiments.

- Two virtual reality systems, one developed at the Swedish Institute for Computer Science under the direction of Steven Pink, and the other at the University of North Carolina under Don Smith.

- A distributed rendering system used in a parallel graphics application.

The applications section of the book brings together a sampling of papers from these areas.

1.7 Underlying Theory

A different perspective on Isis derives from the theoretical community, which has a long-term interest in reliability and which introduced many of the basic concepts on which Isis builds. These include distributed agreement, atomic broadcasting, transactions and transactional serializability, and knowledge in distributed systems.

During the early 1980s, a great deal of activity focused on what might be called the *methodological foundations* of distributed computing. This work was characterized by its emphasis on the importance of abstraction as a tool in distributed systems.

For example, one of the earliest distributed protocols to receive any substantial degree of attention was concerned with solving the *consensus* problem. In this problem, a set of processes in a network reach a point in their execution where it becomes necessary to agree upon an action. Perhaps some external stimulus triggers this situation: the precise question of why the consensus is needed is rarely viewed as a key aspect of the problem. At any rate, the processes must exchange messages until a consensus value is decided upon, at which

point the problem can be viewed as having been solved. Consensus abstracts a type of basic interaction that real distributed systems, such as Isis, must solve repeatedly.

A number of factors make distributed consensus difficult. These include the various kinds of failures that can occur, ranging from benign failures like message loss and process or machine crashes, to more annoying ones, such as message duplication, out-of-order message delivery, transient failures (a process may be unreachable for a period of time, but then recover suddenly), and even malicious failures like message corruption, outright attacks on a set of processes by an aggressive intruder, and so forth. Beyond failure, one must address issues stemming from concurrency, from the lack of any agreed upon source of time (most machines have private clocks that can drift), and problems such as communication partitioning, which leaves islands of nodes able to communicate only within themselves, cut off from other islands. The theory community set out to systematically understand when the consensus problem can be solved at all, and how the solution changes with the failure model.

To understand the limits of what can be done, the earliest solutions to the consensus problem focused on a very simplistic model of the network – a *synchronous* model, in which programs execute in lock-step rounds, each exchanging a message with all others during each round. Needless to say, no real network operates this way. But, the argument goes, if a problem turns out to be impossible *even in this simplified setting*, then surely it is also impossible in a more demanding setting such as a real distributed environment. The answer is a mixed one: so-called crash failures, where a process simply halts, are easy to deal with in the synchronous model. More severe failure models, and especially the malicious models in which some processes may behave corruptly, are very difficult to overcome, requiring many rounds of message exchange and large numbers of messages. (If one is willing to settle for a probabilistic solution, the results are more encouraging.)

The situation for arbitrarily asynchronous systems is also characterized by a mixture of negative and positive results. The best known negative result is the impossibility proof of Fisher, Lynch, and Patterson [FLP85]. They showed that even in simple *crash failure* models, if the network is completely asynchronous there will be ways to prevent a system from reaching consensus when even a single process may fail during the consensus protocol. It is important to stress that this is not at all the same thing as saying that consensus can never be reached in asynchronous systems–actually, the situations that can prevent a system from reaching consensus are fairly complex, and hence not very probable. What this result tells us is that real distributed systems will sometimes run into situations where at least some of their nodes would have to stop and wait for other nodes to recover. We can try and minimize the frequency of such events, or we can try to build systems that don't need to solve problems that are as hard (abstractly) as consensus, but to the extent that the latter approach is impractical, we cannot completely evade the occasional need to wait.

At the other extreme is the so-called *failstop* model, in which faulty processors and processes are required to stop and all interested processes can detect this. Proposed by Schlichting and Schneider [SS83], this model yields a much simpler programming environment. A second model was proposed by Lamport, and is known as the *state machine* approach to fault tolerance. In this approach, groups of programs executing identical code are used to overcome various sorts of failures.

Isis turns out to be in the class of systems that can be used to solve consensus, and hence are subject to the FLP restrictions. The particular manifestation of these restrictions in Isis is that certain partition failures, namely those in which neither partition contains a majority of the set of previously operational sites, can prevent Isis from committing new membership lists for the system, which might cause the system to come to a halt until the partition is fixed. For example, if 15 sites are running Isis and a network failure suddenly splits this system into three groups of 5 sites each, Isis will shut down in all three partitions until communication is restored to the degree that a partition with 8 sites can be constituted. Chandra has explored the FLP result in some detail, and has determined the exact situation in which progress cannot be made, and Aleta Ricciardi has looked at the Isis group membership problem (GMP) in the abstract, developing an optimal solution subject to the FLP/Chandra constraints.

One can view Isis as using a form of the failstop model, but rather than simply assuming that failures are failstop ones, the model is implemented over weaker assumptions using Ricciardi's GMP service. Moreover, one can view Isis groups as implementing a sort of optimized version of the state machine approach. Although neither analogy is exact, the relationship between Isis and the prior body of theory is substantial.

Similarly, theoretical research provides some insight into the atomic multicast problem. In this problem, a sender process wishes to send a message to a set of processes in such a way that the message either gets to all processes, or (and only if failures occur) the dust settles and no surviving process has received the message at all. In the standard version of the problem, one also requires that if two messages are sent concurrently (even by independent senders), the delivery order used is the same for all destination processes. The resulting protocol is often called *abcast.*

We will not discuss atomic broadcasting in Byzantine settings–a problem closely related to consensus. Rather, we point here to a number of more pragmatically oriented papers, often trying to use very realistic network models, and sometimes seeking to exploit special hardware features such as broadcast. Although our own work on Isis represented a contribution in this area, prior to anything of ours a number of atomic multicast protocols had been proposed, such as the token-ring protocol of Chang and Maxemchuk [CM84], the reliable multicast protocol of Schneider, Gries, and Schlichting [SGS84], and the State Machines protocol of Lamport and Schneider [Sch90]. The Isis version of abcast is actually implemented in two ways within the system, as will be clear to readers of the original papers, but our work is shifting to the more modern protocols developed by Birman, Schiper, and Stephenson in 1991 [BSS91]. Other systems that implement abcast protocols include the Delta-4 work of Powell and Veríssimo [Pow91], Melliar-Smith's Total system [MSMA90], Kaashoek's abcast protocol for Amoeba [KTFHB89], and Amir, Dolev, and Malki's work on Transis [ADKM92b]. A closely related body of work focuses on real-time properties, and includes protocols such as the Delta-T protocols of Cristian *et. al.* [CDSA90].

The abcast protocol is inherently costly in an important way. Consider two processes, A and B, that concurrently communicate to a process group G of which both are members. This is a common architecture in systems like Isis: it lets the group members maintain a replicated data object by atomically broadcasting updates, which all members apply to their local replica in the same order. Coupled with a way of obtaining the *state* of the data

object when a process first joins the group (a mechanism provided in Isis), such replication can be employed for a great variety of applications. Notice that one or both of A and B will have to wait before seeing its own message. The reason for this is that the message ordering that abcast will use is not predictable in advance. In a run where A was the only process sending to G, A's message would clearly be the next one delivered. In a run where B's message was alone, that message would be the next. But, without knowledge of the existence (or nonexistence) of other senders, the abcast protocol is compelled to check for other senders, pick an ordering, and get this information back to the sender as well as the other destinations. Thus, any abcast protocol will block some messages while some sort of a protocol runs. This process can be slow.

The situation for abcast can be contrasted with the situation for *fbcast*, a FIFO protocol that delivers messages in the order in which they were sent, provided that the messages had the same sender. In our example, A and B are distinct processes, hence fbcast would not guarantee any particular delivery ordering at all. In particular, the outcome in which some processes see A's message first and some see B's message first would be allowed–and not even hard to provoke. One would need to give careful thought to a system before replacing an invocation of abcast with an invocation of fbcast: although some applications might not be sensitive to the relative order in which messages arrive within a group, it is easy to believe that many applications would be highly sensitive to this type of message delivery ordering. Nonetheless, fbcast has a major advantage over abcast: it does not delay messages in the way seen above. fbcast may delay a message if a prior one from the same sender is somehow lost, but the sender can always deliver a copy of the message to itself without delay, and in most cases, other processes can deliver an fbcast without delay.

This leads to an important feature of Isis. Early in our work on Isis, we proposed that the system should support a protocol called *cbcast*, which is like fbcast but has been extended to also detect cases where two messages were sent in some order by an application that spans multiple processes. For example, fbcast would not order messages m and m' if A sent m, the reception of which caused B to send m'. With fbcast, some third process C could receive m' first, and then m, even though m' may make sense only after m has been received.

In 1987, our initial Isis papers showed that this type of ordering guarantee can be provided at almost the same cost as fbcast. The cbcast protocol is like an fbcast protocol, but interprets *first* using a definition proposed by Lamport in 1978 [Lam78]. Schmuck showed that abcast and cbcast are each *complete* for a class of protocols (see Paper 14 and [Sch88]), in the sense that these protocols have inherently different costs and that each can emulate the behavior of any other protocol in their class, at the same cost (in terms of message counts). By 1991, we had demonstrated that, at least in bursty communication settings, cbcast can actually have the same cost as fbcast.

The broader importance of cbcast is a matter of ongoing debate. Within Isis, the performance impact is substantial, and we continue to believe that this protocol is the appropriate one on which to base a distributed system (with abcast built as an algorithm over cbcast, although implemented in the communications library). But Isis is a fairly specialized system, and this leaves open the question of whether causal delivery ordering should be supported in other settings too. In some of our work we have argued in favor this

view. Other members of the research community dispute this claim. Some favor supporting only abcast, and others (often citing the end-to-end argument), argue against any form of system-imposed ordering at all.

Looking beyond this issue, a new question is emerging. Systems like Isis and Transis are able to support a form of consistent distributed behavior in which programs trust one another to take actions, and are safe in so doing. In particular, such a program is guaranteed to be correct in its belief (assuming the programs themselves are correct!), to later discover that some other program crashed before taking a desired action, or to crash itself. In the absence of failures, these systems allow consistent distributed actions and states to be maintained. In contrast, most distributed systems lack any notion of distributed consistency, and it is not straightforward to layer such a property over a system that does not support it. The deep question all this raises is how consistency, the ability to solve consensus, and solutions to problems like the group membership problem relate to one another. It seems safe to predict that active work in this area will continue for some time.

1.8 The Horus Architecture

Although group communication systems have become popular, software support for such computing remains complex and poorly integrated with modern operating systems structures. This motivated us to redesign Isis, an activity that resulted in the Horus system, which brings microkernel design techniques to bear on the problem. Horus is lightweight and fast, making it well suited for embedding into operating systems like Mach and Chorus, and is flexible enough to serve as a base for experiments in high availability security technologies, real-time communication, and control of high speed communication devices.

Although Horus is still in a early stage of development, our initial experience with the system has been very positive. First, Horus confirms that high performance group communication can benefit from the same sorts of modularity techniques and optimizations as are used to enhance the performance of RPC and other communication methods. Second, Horus shows that efficient group communication support can exist side by side with other forms of IPC, without impacting on IPC performance. Third, Horus is easily reconfigured to change the properties of the group communication protocols, or to employ an alternative implementation of some of its modules. This permits the system to be used as a base for experiments on systems requiring somewhat different functions than the defaults for Horus.

Having completed the basic system, we are currently exploring a number of applications for which we believe it to be particularly well suited. One of these is to support parallel computing through interfaces such as are used in the popular pvm system, or the emerging mpi standard. In these applications, Horus would run over a very high speed lower level message passing technology, such as the *Active Messages* model developed by Culler and Von Eicken. Over Horus, one would run applications directly on the message passing interfaces, or would use a compiler to convert from some sort of shared memory single–program image into a message-passing distributed application.

A second exciting direction is to use Horus over advanced networking technology, such as ATM. In this approach, Horus would be responsible for managing and controlling ATM connections, setting quality of service parameters, and assisting the applications programmer

in orchestrating data flow through the complex lower layers of a typical ATM protocol stack and switch.

Yet a third direction involves the integration of Horus with a real-time protocol suite. In this work, called Corto, we are extending the basic Horus support with additional protocols that deal with issues such as priority-based scheduling and guaranteeing real-time time limits and synchronization. All of this work is underway as we write this introduction in 1993, and will be reported in future papers.

1.9 A Security Architecture For Horus

Much of our early experience with Horus results from work on a security architecture. There exists considerable experience with individually addressing the needs for security and fault tolerance in distributed systems. However, much less is understood about how to simultaneously address these needs in a single, integrated solution. Indeed, the goals of security and availability have traditionally been viewed as being in conflict, because the only generally feasible technique for making data and services highly available, namely replicating them, also makes them inherently harder to protect.

In our work on this problem, we developed and implemented a security architecture for fault-tolerant systems, which illustrates that this conflict need not result in an unreliable or insecure system. The architecture supports process groups as its primary security abstraction, and provides tools for the construction of applications that can tolerate both benign component failures and advanced malicious attacks. We have integrated this architecture into the Horus system, thereby securing its *virtually synchronous* process group abstraction and the programming tools that it embodies.

The tradeoff between security and availability is addressed in two ways in our architecture. At the level of user applications, the architecture provides a framework that enables the user to balance this tradeoff for each application individually. The framework consists of *secure process groups* within which a user can efficiently replicate applications in a protected fashion. Authentication and access control mechanisms enable the group members to prevent untrusted processes from joining. Provided that they admit only processes on trustworthy sites, the members will enjoy secure communication and correct group semantics among themselves. Thus, the conflict between security and availability at this level of the system is passed to the user by enabling him or her to control where and how widely each application is replicated.

The second level at which this conflict is addressed is in the core security services that underlie these abstractions, and indeed the security of *all* process groups. As do other security architectures, ours uses cryptography to protect communication, and this in turn requires that there exist a secure means of key distribution. Most key distribution mechanisms employ trusted services whose corruption or failure could result in security breaches or could prevent principals from establishing secure communication; it is in these services that the conflict between security and availability is most apparent. We have developed an approach to reconciling this conflict that exploits the semantics of these services and novel replication techniques to achieve secure, fault tolerant key distribution.

1.10 Future Directions

As we assemble this book in 1993, the scope and scale of our activity are more ambitious than at any time in the past. Looking to the future, we are finding ways to apply Isis in demanding and *mission-critical* settings that were out of reach to us a few years ago. A continuous effort to improve the quality and performance of the code has paid off in steadily increasing reliability of the overall system, while a substantial investment in performance improvement has pushed Isis close to the technology limits for applications layered over Unix. With Horus, we are taking on ATM applications, parallel computing applications, and beyond that, embedded applications of the sort that will become common as computing becomes more ubiquitous. The development of a comprehensive theory of consistency in asynchronous distributed environments now appears to be within reach. Such a theory would answer many questions about the fundamental nature of distributed systems that guarantee reliability.

High speed communication and computing are transforming our world. Technology is advancing at a fearful pace, and with it new businesses, new styles of doing business, and new lifestyles for individual consumers are emerging. The computer software that supports of all this will need the sorts of powerful tools that Isis and Horus embody: the solutions of the past, while adequate for the needs of the period, are not up to the challenges of massively large scale, critical applications. Thus the opportunity is clear. We look to a bright future for reliable distributed computing systems, and with our colleagues and the users of Isis and Horus, see this book as merely the start of what could become a long story.

1.11 Outline of this Book

This book consists of four parts. In Part 1, Fundamentals, we have collected papers that deal with the underlying ideas of the Isis system. The second Part, Redesign, contains new papers that describe Horus, a system based on our experience with Isis. Part 3 Protocols, consists of papers that describe the protocols that are used in Isis and Horus. The last Part, Tools and Applications, contains user-contributed papers on Isis applications.

1.12 Acknowledgements

The Isis project began in 1983 under the direction of Kenneth P. Birman and several students – Tommy Joseph, Thomas Raeuchle, Wally Deitrich, and Amr El Abbadi. During the subsequent ten-year period, a huge number of people have contributed in varied ways to our work. Omitting authors of papers included in this text, these include: Yair Amir, Özalp Babaoğlu, Rod Bark, Micah Beck, Navin Budhiraja, Tushar Chandra, Mani Chandy, Tim Clark, Eric Cooper, Flaviu Cristan, Rich Draves, Yves Eychenne, Holger Herzog, Sue Honig, Yu-Jen Hsiao, Guerney Hunt, Bill Joy, Frans Kaashoek, Mike Kalantar, Ken Kane, Jacob Levy, Messac Makpangou, Dalia Malki, Jay Misra, Shivakant Misra, Shigeki Morimoto, Jishnu Mukerji, Gil Neiger, Doug Orr, Richard Platek, Franklin Reynolds, Mark Rozier, Satya, André Schiper, Fred Schneider, Karen Seymour, Stefan Sharkansky, Dennis Shasha,

Alex Siegel, Dale Skeen, Don Smith, Alfred Spector, Mark Steiglitz, Joe Sventek, Gautam Thaker, Sam Toueg, Paulo Veríssimo, Ellen Vorhees, Chung Wang, John Warne, Dan Wolfson, Bill Weihl, Raphael Yahalom, and Raffi Yaholom. Of course, even this extensive list is only partial, and we extend our thanks as well to all of those who contributed in many ways, large and small, to the success of the effort.

We are also grateful for the steady funding and encouragement we have received from ARPA, NASA, ONR, and NSF, and for the generosity of our corporate sponsors, notably IBM, GTE, HP, and Siemens. Visitors and support from Hitachi, Mitsubishi, and Sony have also been of great value to us in pursuing our research.

Finally, special thanks are due to Maureen Robinson and Cindy Williams, without whose help this manuscript could not have been produced, and who produced the great majority of figures in this text.