BRAN SELIC

Distributed Software Design: Challenges and Solutions

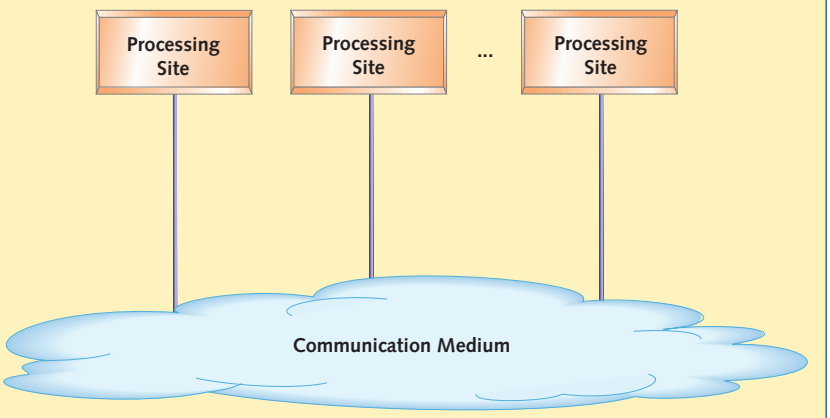
In contrast to centralized systems, distributed software systems add a new layer of complexity to the already difficult problem of software design. In spite of that and for a variety of reasons, more and more modern-day software systems are distributed.

In some cases, such as telecommunications systems, distribution is inherent in the problem domain and cannot be avoided. In other cases, we rely on physical distribution to achieve properties that cannot be effectively realized with centralized systems. For instance, we may want to use multiple physically distributed processors to achieve a high degree of availability. Thus, if one processor experiences a hard failure, another one can take over its functionality. Another reason for distribution is performance. By splitting the processing load across multiple processors, it is generally possible to achieve higher throughput and faster response than with just a single processor.

We define a distributed software system (see Figure 1) as *a system with two or more independent processing sites that communicate with each other over a medium whose transmission delays may exceed the time between successive state changes.* Note that this definition is broad enough to encompass logically distributed systems. These are software systems that are concentrated on a single processing node but, for one reason or another,[1] exhibit the above properties. An example of such a system is one that spans multiple heavyweight processes running on the same processor and which interact by exchanging asynchronous messages. Since each process is an independent unit of failure, it fits our definition. (For this reason, our term *distributed site* should not be taken necessarily as a synonym for the term "processing node.") While logically distributed systems cannot achieve the full benefits of physically distributed systems, the problems that they present to developers are practically the same.

One of the central problems with unreliable communication media is that it is not always possible to positively ascertain that a message that was sent has actually been received by the intended remote destination.

FIGURE 1 Distributed system architecture



The challenges of distributed software

The majority of problems associated with distributed systems pertain to failures of some kind. These are generally manifestations of the unpredictable, asynchronous, and highly diverse nature of the physical world. In other words, because fault-tolerant distributed systems must contend with the complexity of the physical world, they are inherently complex.

Failures, faults, and errors

Let's introduce some basic terminology.[2] A *failure* is an event that occurs when a component fails to behave according to its specification. This is usually because the system experiencing the failure has reached some invalid state. We refer to such an undesirable state as an *error*. The underlying cause of an error is called a *fault*.

For example, a bit in memory that is stuck at "high" is a fault. This will result in an error when a "low" value is written to that bit. When the value of that bit is read and used in a calculation, the outcome will be a failure.

Of course, this classification is a relative one. A fault is typically a failure at some lower level of abstraction (that is, the stuck-high bit may be the result of a lower-level fault due to an impurity in the manufacturing process).

When a failure occurs, it is first necessary to detect it and then to perform some basic failure handling. The latter involves diagnosis (determining the underlying cause of the fault), fault removal, and failure recovery. Each of these activities can be quite complex.

Consider, for example, failure diagnosis. A single fault can often lead to many errors and many different cascading failures, each of which may be reported independently. A key difficulty lies in sorting through the possible flurry of consequent error reports, correlating them, and determining the basic underlying cause (the fault).

Processing site failures

Because the processing sites of a distributed system are independent of each other, they are independent points of failure. While this is an advantage from the viewpoint of the user of the system, it presents a complex problem for developers. In a centralized system, the failure of a processing site implies the failure of all the software. In contrast, in a fault-tolerant distributed system, a *processing site failure* means that the software on the remaining sites needs to detect and handle that failure in some way. This may involve redistributing the functionality from the failed site to other, operational, sites, or it may mean switching to some emergency mode of operation.

Communication media failures

Another kind of failure that is inherent in most distributed systems comes from the communication medium. The most obvious, of course, is a complete hard failure of the entire medium, whereby communication between processing sites is not possible. In the most severe cases, this type of failure can lead to partitioning of the system into multiple parts that are completely isolated from each other. The danger here is that the different parts will undertake conflicting activities.

A different type of media failure is an intermittent failure. These are failures whereby messages travelling through a communication medium are lost, reordered, or duplicated. Note that these are not always due to hardware failures. For example, a message may be lost because the system may have temporarily run out of memory for buffering it. Message reordering may occur due to successive messages taking different paths through the communication medium. If the delays incurred on these paths are different, they may overtake each other. Duplication can occur in a number of ways. For instance, it may result from a retransmission due to an erroneous conclusion that the original message was lost in transit.

One of the central problems with unreliable communication media is that it is not always possible to positively ascertain that a message that was sent has actually been received by the intended remote destination. A common technique for dealing with this is to use some type of *positive acknowl-*

However, even if the transmission delay is constant, the problem of out-of-date information still exists.

FIGURE 2 Relativistic effect

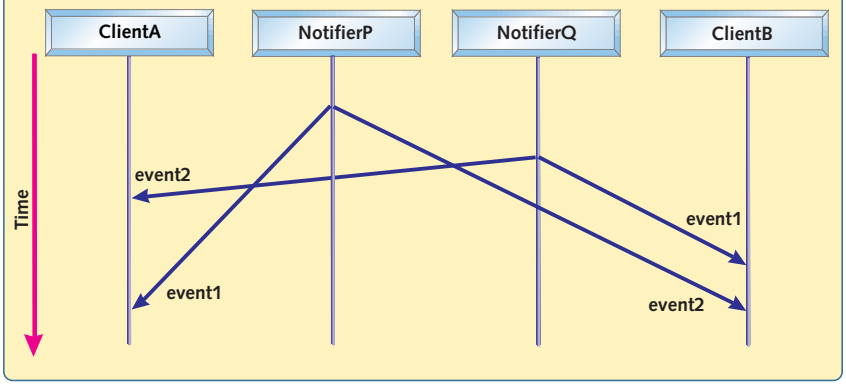
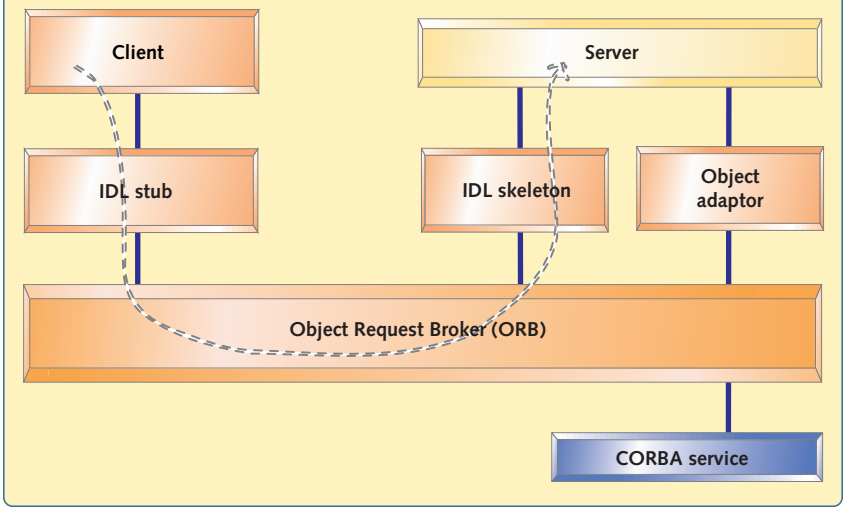


FIGURE 3 CORBA components and model



edgement protocol. In such protocols, the receiver notifies the sender when it receives a message. Of course, there is the possibility that the acknowledgement message itself will be lost, so that such protocols are merely an optimization and not a solution.

The most common technique for detecting lost messages is based on time-outs. Namely, if we do not get a positive acknowledgement that our message was received within some reasonable time interval, we conclude that it was dropped somewhere along the way. The difficulty of this approach is distinguishing whether a message (or its acknowledgement) is simply slow or actually lost. If we make the time-out interval too short, we risk duplicating messages and, in some cases, reordering. If we make the interval too long, the system may become unresponsive.

Transmission delays

While transmission delays are not necessarily failures, they can certainly lead to failures. We've already discussed the situation where a delay can be misconstrued as a lost message.

Two different types of problems are caused by message delays. One type of problem results from *variable* delays (jitter). That is, the time taken for messages to reach the destination may vary significantly. The delays depend on a number of factors, such as the route taken through the communication medium, congestion in the medium, congestion at the processing sites (for example, a busy receiver), intermittent hardware failures, and so on. If the transmission delay were constant, we could better assess when a message has been lost. For this reason, some communication networks are designed as synchronous networks, so that delay values are fixed and known in advance.

However, even if the transmission delay is constant, the problem of *out-of-date information* still exists. Since messages convey information about state changes between components of the distributed system, the information in these messages may be out of date if the delays experienced are greater than the time required to change from one state to the next. This can lead to unstable systems. Imagine trying to drive a car in a situation where the visual input to your eyes is delayed by several seconds.

Transmission delays also lead to a complex situation that we will refer to as the *relativistic effect*. This is because transmission delays between different processing sites in a distributed system may be different. As a result, different sites could see the same set of messages but in a different order.

In Figure 2 case, distributed sites `NotifierP` and `NotifierQ` each send out a notification about an event to the two clients (`ClientA` and `ClientB`). Due to the different routes taken by the individual messages and the different delays along those routes, we see that `ClientB` sees one sequence (`event1` followed by `event2`), whereas `ClientA` sees another. As a consequence, the two clients may reach different conclusions about the state of the system.

Consider the case of two army generals of ancient times, when communication between distant sites could only be achieved by physically carrying messages between the sites.

Note that the mismatch here is not the result of message overtaking (although this effect is compounded if overtaking occurs), but is merely a consequence of the different locations of the distributed agents relative to each other.

Distributed agreement problems

The various failure scenarios in distributed systems, and transmission delays in particular, have instigated important work on the foundations of distributed software. Much of this work has focussed on the central issue of *distributed agreement*. There are many variations of this problem, including time synchronization, consistent distributed state, distributed mutual exclusion, distributed transaction commit, distributed termination, distributed election, and so on. However, all of these reduce to the common problem of reaching agreement in a distributed environment in the presence of failures.

We introduce this problem with the following apocryphal story. Consider the case of two army generals of ancient times, when communication between distant sites could only be achieved by physically carrying messages between the sites. The two generals, Amphiloheus and Basileus, are in a predicament whereby the enemy host lies between them. While neither has the strength to single-handedly defeat the enemy, their combined force is sufficient. Thus, they must commence their attack at precisely the same time or they risk being defeated in turn. Their problem then is to agree on a time of attack.

Unfortunately for them, a messenger going from one general to the other must pass through enemy lines with a high likelihood of being caught.

Assume then, that Amphiloheus sends his messenger to Basileus with a proposed time of attack. To ensure that the message was received, he demands that the messenger return with a confirmation from Basileus. (While this is going on, Basileus could be in the process of sending his own messenger with his proposal—possibly different—for the time of the attack.)

The problem is obvious: if the messenger fails to get back to Amphiloheus, what conclusions can be reached? If the messenger succeeded in reaching the other side but was intercepted on the way back, there is a possibility that Basileus will

It has been formally proven that it is not possible to guarantee that two or more distributed sites will reach agreement in finite time over an asynchronous communication medium, if the medium between them is lossy or if one of the distributed sites can fail.

attack at the proposed time but not Amphiloheus (since he did not get a confirmation). However, if the messenger was caught before he reached Basileus, then Amphiloheus is in danger of acting alone and suffering defeat. Furthermore, even if the messenger succeeds in getting back to Amphiloheus, there is still a possibility that Basileus will not attack, because he is unsure that his confirmation actually got through. To remedy this, Basileus may decide to send his own messenger to Amphiloheus to ensure that his confirmation got through. But, the only way he can be certain of that is if he gets a confirmation of his confirmation. Since there is a possibility that neither messenger got through to Amphiloheus, Basileus is no better off than before if his second messenger does not return.

Clearly, while sending additional messengers can increase the likelihood that a confirmation will get through, it does not fundamentally solve the problem since there will *always* be a finite probability that messengers will get intercepted. It's a case of "does he know that I know that he knows that...?" and so on.

Impossibility result

Our parable of the generals is simply an illustration of a fundamental impossibility result. Namely, it has been formally proven that it is not possible to guarantee that two or more distributed sites will reach agreement *in finite time* over an asynchronous[3] communication medium, if the medium between them is lossy [7] or if one of the distributed sites can fail [2].

This important result is, unfortunately, little known and many distributed system developers are still trying to solve what is known to be an unsolvable problem—the modern-day equivalent of trying to square the circle or devise a perpetual motion machine. The best that we can do in these circumstances is to reduce the possibility of non-termination to something that is highly unlikely.

The Byzantine generals problem

A common paradigm for a particular form of the distributed agreement problem is the so-called *Byzantine generals problem.*[4] In this problem it is assumed that at least one processing site is faulty. Furthermore, any faulty processing sites are assumed to be malicious in the sense that they are trying to subvert agreement by intentionally sending incorrect information. While the number of such sites is known, their identity is not. The objective is to find an algorithm whereby all the non-faulty sites can reach agreement despite the disruptive actions of the faulty sites.

This may seem like a rather exotic and needlessly paranoid problem of little practical value. However, it is actually quite important for fault-tolerant systems. The point is that any solution that can survive a malicious attack will be robust enough to survive practically any type of fault.

Heterogeneity

Many distributed systems arose out of the need to integrate legacy standalone software systems into a larger more comprehensive system. Since the individual systems were often developed independently of each other, they may have been based on different architectural principles and even different programming and operating system technologies. This creates another common problem of distribution: integration of heterogeneous technologies. For example, one system might use a 16-bit integer format with the most significant bit in the leftmost position, while its peer may use a 32-bit format with the most significant bit in the rightmost position.

System establishment

Since there are multiple processing sites in a distributed system, there are multiple loci of control. Note that, in general, it cannot be assumed that all the sites will necessarily start executing at the same time. For example, a site that failed and was then recovered will come up long after all the other components are all operational. A major problem is how these distributed sites find and synchronize with each other.

Distributed algorithms and techniques

In this section we examine some of the most common distributed algorithms. By "distributed" we mean that the decisions are reached by consensus, with each site making its own decision according to the rules of the algorithm and the evidence presented by the other participating sites. These algorithms generally are based on shared (common) knowledge [7]; that is, each site must know what all the other sites know. Consequently, an important related problem is how to disseminate distributed knowledge reliably and efficiently.

A major difficulty with common knowledge is that all parties have to know about each other. In dynamic systems, where sites may come and go, this can add a significant amount of overhead. In particular, difficulties result when a new site joins while a distributed algorithm is in progress. In those cases, it is typical to refuse entry to the newcomer until the algorithm terminates. Conversely, a site may fail in the middle of executing a distributed algorithm, which means that all the other participants must be notified.

Communication techniques

Both synchronous and asynchronous communication is used in distributed systems. Synchronous communica-

In the Byzantine generals problem, the objective is for all the non-faulty sites to agree on a value, even if faulty sites are sending incorrect information.

tion, in the form of remote procedure call or rendezvous, has the advantage of combining synchronization with communication. Once a synchronous invocation completes, the invoking party knows positively whether the communication has succeeded and, if it has, it also knows something about the state of the invoked site. For this reason, synchronous distributed communication primitives are directly supported in many distributed languages such as Ada [10] and occam [12].

The principle is simple. In a rendezvous, for instance, one site performs a remote invocation while the receiver executes a "receive" operation. Unless the two operations are executed simultaneously, one of the parties will have to wait for the other (which is why it is called a "rendezvous"). When the rendezvous occurs, the invoking site is suspended until the invoked operation is completed at the remote site and a reply is received. The receiving site simply continues its execution.

Unfortunately, synchronous communication typically involves significant overhead (mainly because of the need to deal with failures) that reduces efficiency. Furthermore, while a synchronous communication is in progress, both participants are unreceptive to any other communications. In many reactive and real-time systems, such prolonged periods of autism are unacceptable.

In contrast, asynchronous communication, or message passing, is simpler and more efficient, but does leave the problem of synchronization to the application. Messages are simply sent to the receiver regardless of whether the receiver is ready to accept them.

The Byzantine generals algorithm

In the Byzantine generals problem, the objective is for all the non-faulty sites to agree on a value, even if faulty sites are sending incorrect information. Furthermore, we require that the agreed on value must be based on the information received from all other non-faulty sites. The latter is stipulated to prevent agreement on an arbitrary value but to represent a true consensus.

The problem and a solution were first described by Lamport, Shostak, and Pease.[9] We assume that the following special conditions hold:

All non-faulty sites will correctly relay the value they received from the coordinator. Faulty sites, on the other hand, are free to do as they choose, including sending the wrong value or not sending a value at all.

- The communication medium is reliable (that is, no possibility of failures such as loss, duplication, or reordering of messages)
- The absence of a message can be detected
- The receiver of a message knows unequivocally the identity of the sender (that is, the sender's identity cannot be forged)
- It is possible to positively determine that a message was not sent by a particular site (as opposed to being lost in the medium)

Clearly, considering our previous discussions on the difficulty of achieving these conditions in distributed systems, these are rather significant constraints on the applicability of the algorithm. They are introduced to simplify the problem. Variants of the algorithm have been developed that relax some of these conditions.

In the Lamport-Shostak-Pease algorithm, one of the sites initiates the agreement process. We refer to this site as the coordinator while the other sites are called followers. The algorithm generally proceeds as follows:

- The coordinator sends its value to each of the followers
- Each follower relays the value that it receives from the coordinator to all other followers (but not the coordinator). If no value was received from the coordinator, a shared default value is used instead. In effect, the follower acts as a coordinator for the reduced set of sites that excludes the coordinator of the previous round. Note that this "relaying" step is performed in parallel by each of the followers. This means that there can be multiple rounds of relaying, until each site finds itself in a situation where it has no followers left
- Each follower chooses the majority value, or, if no values are received at all, it chooses the shared default value. (In case of ties, each site invokes the same method for breaking the tie, thereby guaranteeing agreement)

All non-faulty sites will correctly relay the value they received from the coordinator. Faulty sites, on the other hand, are free to do as they choose, including sending the wrong value or not sending a value at all.

Note that, if there is no majority, the algorithm cannot work. In fact, it can be shown that, in the case of n faulty sites, the algorithm only works if there are at least $(3n+1)$ sites. This means that the algorithm can only work if there are at least four sites. (A variant of the algorithm in which faulty followers are constrained to correctly relay the coordinator's value only requires $[2n+1]$ sites.)

This algorithm works even if the coordinator is faulty, because it guarantees that all non-faulty sites will agree on the same value. Further details, including a proof of correctness of the algorithm, can be found in Lamport, Shostak, and Pease.

Distributed mutual exclusion

The basic principle here is that a site wanting to enter a critical section must inform all the other sites and then wait until it has been given approval by all other sites. A request for entering a critical section is broadcast reliably to all the other sites. The request contains the identity of the requesting site as well as a timestamp of the request.[6] The timestamp is required to ensure fairness.

When a site S receives a request from site R to enter a critical section, it will either give its permission immediately or defer the reply until later. The algorithm that it follows is:

- If S is already in a critical section, it delays sending its permission
- If S is not waiting to enter a critical section, it replies immediately granting permission
- If S is itself waiting to enter a critical section, it compares the timestamp of its own request with the timestamp of R's request and, if the latter is earlier, it replies immediately with its permission. However, if S's timestamp is earlier, then it postpones its permission
- Postponed permissions are granted as soon as S exits its critical section
- A site does not enter its critical section until all other sites have given it permission

Distributed election

A common problem that occurs in certain kinds of distributed systems is the *distributed election problem*. In this case, it is necessary to elect a *leader* among a set of peers. The peers are symmetric in the sense that they are all equally capable of being leaders, but only one is required to perform the functions of the leader. The advantage of this scheme is that if a leader fails, the remaining sites can simply undertake another election and select a new leader. This avoids a single point of failure and increases the potential system availability.

A widely used election algorithm is called the Bully algorithm [3]: a site entering the election procedure first sends its bid to all other sites. The bid is typically based on some quality that is unique to each site, such as a site identifier, so that there are no duplicate bids. If a leader already exists, it responds to the new site and the new site simply enters the *monitoring* state. In this state, the site monitors the operational status of the elected leader (for example, through a periodic poll).

If no leader exists, the candidate site waits to receive the bids of all other candidates that it receives in response to its own bid. Once these are all in, each site evaluates the bids.

If failures occur during the broadcast, the algorithms will fail since the participants will have inconsistent views of the system state.

The site with the highest bid is selected as the leader (which explains the name of the algorithm). This site sends a confirmation message to all other sites.

If a site in the monitoring state determines that the current leader has failed, it re-enters the election procedure by sending its bid to all sites that had previously entered a higher bid than its own. If none of these respond, the site elects itself as the leader and informs all lower-bid sites. However, should a higher-bidding site respond, it waits to receive a confirmation message from the new leader. If such a message does not arrive, the election process is re-entered.

This simple sounding algorithm is fraught with practical implementation difficulties. As sites come and go, it is difficult to keep track of who is present and what phase of the election process they are in. If the communication medium is not perfectly reliable, the difficulties are often insurmountable. Detection of leader failures can also be problematic if the polling is based on timeouts.

Fault-tolerant distributed broadcasting

As we have seen, a key element of many distributed algorithms is the need to reliably broadcast local information to all the participants in an algorithm. If failures occur during the broadcast, the algorithms will fail since the participants will have inconsistent views of the system state. For this reason, it is very useful to provide *fault-tolerant broadcast services* in a distributed system.[5]

The weakest form of fault-tolerant broadcast is called a *reliable broadcast*. Within a given group of sites, it guarantees three basic properties:

- All messages sent are received by all non-faulty sites in the group
- All non-faulty sites in the group will receive the same set of messages
- Only sent messages are delivered (that is, there are no "spontaneous" messages)

Note that reliable broadcast does not guarantee the order in which the messages will be received. The order of reception can be crucial in some cases. For example, if an "on" message is sent after an "off" message, it is important that they are received in that order. For this reason a number of other types of fault-tolerant broadcasts are defined with progressively stronger restrictions on the delivery order of messages.

A *FIFO broadcast* is a reliable broadcast that guarantees that messages sent by the same sender are delivered in sending order. Note, however, that multiple sites could be broadcasting at the same time so that the possibility exists even with FIFO broadcasts that different members of the group will receive messages from *different* sites in a different order. For this reason, we define *atomic broadcasts*, which guarantee that all non-faulty sites receive all messages in the same order.

Another type of reliable broadcast is *causal broadcast*. In this case, the delivery order is guaranteed to respect causal relationships. That is, it is not possible for a message to be delivered before another message that precedes it in the system cause-effect chain. Finally, it is sometimes useful to provide a *causal atomic broadcast* that combines the features of causal and atomic broadcasts.

Distributed groups

Distributed algorithms are designed to operate within a given group of participants. Hence, they always imply a scope, a community of distributed sites within which the algorithm applies. Membership in this community may change dynamically as sites join and depart. For example, the fault-tolerant broadcast algorithms described above are designed with this model in mind. This led to the idea of providing operating system support for *distributed groups*.[4] A distributed group is simply a group of processing sites that are joined together for some common purpose. Sites may join and leave the group as they see fit.

These distributed groups provide the basic services required by many different distributed algorithms and applications. A typical group facility provides the following:

- A *membership service*, which is used to track which sites are currently in the group and to inform members of the group when new sites join or leave the group
- *Fault-tolerant broadcast services* (as described previously) as well as reliable point-to-point (member-to-member) communications
- A *mutual exclusion service* that provides a basic distributed locking mechanism within the group

By relieving applications from having to implement the complex machinery involved with these services, a major simplification of applications can be obtained.

Centralized alternatives

An alternative to fully distributed algorithms is to appoint one well-known site as a central arbiter. When agreement needs to be reached, all the other sites simply consult the central arbiter and abide by its decision. This is a common solution in many distributed systems but it has two drawbacks. The first is that a central site represents a potential bottleneck. However, as we have seen previously, distributed algorithms require a significant overhead to establish and maintain common knowledge. In fact, it is generally the case that a centralized algorithm is actually more efficient than a distributed one. A more serious drawback is that a central arbiter becomes a single

point of failure potentially reducing the level of availability that can be achieved in the system.

One approach that can alleviate the latter problem is to define a fault-tolerant arbiter. This can be done by defining a set of processors all equally capable of playing the arbiter role and then electing one to be the active arbiter. Usually, only one level of redundancy is sufficient (one active and one standby), which greatly simplifies the election protocol.[5]

Distributed system technologies

In recent years, a number of pre-packaged distributed software technologies have become available in the marketplace. Perhaps the most widely used are the ones based on the CORBA industry standard maintained by the Object Management Group (OMG) consortium.

CORBA (Common Object Request Broker Architecture) defines a standard for interoperability of distributed applications. At its core is a distributed communication facility, called the *object request broker* (ORB), that allows individual objects to locate and interact with each other.[11] For this reason, the ORB is often referred to as the "object bus." The interacting objects might even be implemented in different programming languages, which means that the systems based on an ORB can be heterogeneous. This is achieved by defining a language-neutral format, called the *interface definition language* (IDL).

IDL is used to define the *interfaces* of remote objects, representing functions that may be invoked by clients and attributes that are publicly accessible. These interface specifications can then be automatically translated into an implementation language by a suitable IDL compiler.

The key components of the CORBA model and its basic operation are described in Figure 3. The *IDL stub* is a local agent of the remote server and has the same interface as the remote server, but is specified in the implementation language of the client. This "native" specification is actually generated by the IDL compiler based on the IDL specification of the server. The *IDL skeleton* encapsulates the server at the remote end and invokes the server on behalf of the client. The *object adapter* provides an interface to the server for accessing the facilities of the ORB as well as other *CORBA services*, such as a name service (used to locate distributed objects during system establishment). These services may be used by the server to realize its function.

The client simply makes a synchronous call via the IDL stub. The stub appears like a simple object that has the appropriate service interface. However, it hides a remote call to the server, executed via the ORB and an up-call through the IDL skeleton (in some cases, the object adapter may also be involved). When the reply information is returned, the client proceeds normally. The entire complex transaction is completely transparent to the client.

There are many more details and variants of this basic scheme, but they are beyond the scope of this article. Interested readers should refer to the OMG spec.[11]

CORBA is just one example of the new generation of technologies and accompanying tools that are becoming available to distributed system developers. Their primary purpose is to hide some of the complexity of distributed systems development by providing standard solutions to common distributed system problems. From that perspective it is useful to characterize the different kinds of "transparencies" (that is, complexity hiding) that such technologies can provide:[8]

- *Access transparency* masks out differences in data representation and the invocation mechanism used to access a service (as in the case of CORBA)
- *Failure transparency* hides the failure and recovery of failed objects from clients through some form of masking redundancy
- *Location transparency* masks out the information about the specific location of some function or service (for example, whether it is local or remote)
- *Relocation transparency* allows an object to be moved to a different location without disturbing the clients of that object
- *Replication transparency* masks out the fact that a server may be redundantly implemented (for performance or availability)
- *Transaction transparency* hides from clients the complex mechanisms and activities required to ensure consistency in a distributed environment.

Summary

There is a common misconception among software practitioners that, with the introduction of the remote procedure call concept, programming of distributed applications becomes pretty much equivalent to more traditional centralized programming. In this article, we demonstrated that this is not the case. Distributed systems introduce a whole new class of phenomena that must be accounted for in program design and implementation. They can be traced ultimately to the physical world: processors fail, communications facilities are unreliable and finite, and so on. The net result is a major increase in the degree of programming difficulty.

We covered some of the basic distributed agreement protocols, commonly encountered in practice, including the Byzantine Generals protocol, distributed mutual exclusion, and distributed election. These all serve to further illustrate the complexity associated with distribution. As pragmatic ways of realizing these types of systems, we discussed some basic implementation techniques such as reliable broadcasts, distributed system groups, and centralized arbiters.

Finally, we described CORBA as an example of the kind of off-the-shelf technologies that are available to modern-day distributed systems designers. Such technologies and related tools can greatly reduce the complexity of distributed system development. **esp**

Bran Selic is principal engineer at Rational Software in Kanata, Ontario, Canada. He has been involved with the development of distributed and fault-tolerant real-time systems for many years and has written a textbook on the subject. Bran is a member of the original team that defined the industry-standard Unified Modeling Language and has recently been working on defining a real-time refinement of this standard. He is also an adjunct professor at Carleton University in Ottawa. You can reach him at bselic@rational.com.

Endnotes

1. One such reason may be that the software is intended to be physically distributed in some future release.
2. This conceptual model is based on the widely adopted work of Randell et al. [13]
3. The case for synchronous media is somewhat different since we can take advantage of the fact that the transmission time of the messages is constant.
4. The name "Byzantine generals" comes from the analogy of a cabal of generals in Byzantium, in which it is known that some of the generals are likely to be traitors to the common cause, but it is not known which ones.
5. There are also variants of the algorithm that do not require a coordinator.
6. The question of how such timestamps are defined is an entire issue unto itself that we will omit in this short overview.

References

[1] Burns, A., and A. Wellings. *Real-Time Systems and Programming Languages* (2nd ed.). Reading, MA: Addison-Wesley, 1997.

[2] Fischer, M., N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, April 1985, p. 374.

[3] Garcia-Molina, H., "Elections in a Distributed Computing System," *IEEE Trans. on Computers*, January 1982, p. 48.

[4] Goyer, P., P. Momtahan, and B. Selic, "A Synchronization Service for Locally Distributed Applications," *Distributed Processing*, North Holland, 1988, p. 3.

[5] Goyer, P., P. Momtahan, and B. Selic, "A Fault-Tolerant Strategy for Hierarchical Control in Distributed Computer Systems," *Proc. 20th IEEE Symp. on Fault-Tolerant Computing Systems (FTCS20)*, IEEE CS Press, 1990.

[6] Hadzilacos, V. and S. Toueg, "Fault-Tolerant Broadcasts and Related Problems," Chapter 5 in S. Mullender (ed.), *Distributed Systems (2nd ed.)*, Reading, MA: Addison-Wesley, 1993, p. 97.

[7] Halpern, J. and Y. Moses, "Knowledge and Common Knowledge in a Distributed Environment," *Proc. of the 3rd ACM Symposium on Principles of Distributed Systems*, 1984, pp.50-61.

[8] International Standards Organization (ISO), Reference Model of Open Distributed Processing (RM-ODP), Part 1: Overview, ISO/IEC 10746-1, May 1995. (also ITU-T standard X.901).

[9] Lamport, L., R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, Vol.4, No. 3, July 1982, pp. 382-401.

[10] Naiditch, D. *Rendezvous with Ada 95, (2nd ed.)*, New York: John Wiley & Sons, 1995.

[11] Object Management Group, The Common Object Request Broker: Architecture and Specification, Revision 2.2, February 1998.

[12] Pountain, D. and D. May. *A Tutorial Introduction to occam Programming*. New York: McGraw-Hill, 1987.

[13] Randell, B., P. Lee, and P. Treleaven, "Reliability in Computing System Design," *ACM Computing Surveys*, Vol. 10, No. 2, June 1978, pp. 123-165.