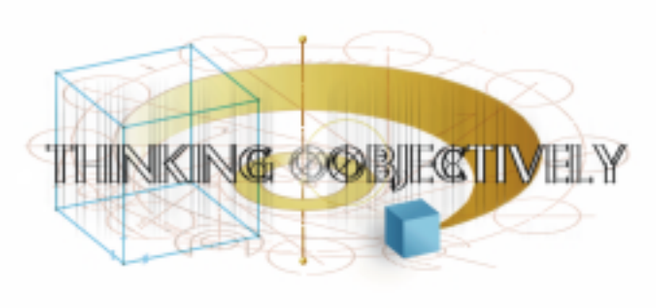

Pascal Felber, Rachid Guerraoui, and Mohamed E. Fayad

Putting OO Distributed Programming to Work

An exercise in abstracting and factoring out failure detection.

We argued in previous columns that object-oriented distributed programming should not be confused with distributed OO programming and that it should not be confused with wrapping existing distributed systems with OO dressing. Taking OO distributed programming seriously entails identifying and classifying the basic abstractions underlying distributed computing.

We attempted to keep our previous claims at an abstract and general level. In this column, we make those claims more concrete. More precisely, we describe OO distributed programming through an exercise consisting of abstracting and factoring out a fundamental component of a distributed system: failure detection.[1]

Adopting a distributed architecture for a given application might be driven by various motivations. One might adopt a distributed solution for an application made of inherently distributed components or decide to distribute a centralized application to take some advantage of distribution. Among the distribution advantages are resource sharing, load balancing, and fault-tolerance. However, these advantages have a dark side, and, to paraphrase L. Lamport, a distributed system is also one that stops you from completing any work because of the crash of a machine you have never heard about.

The notion of partial failure is a fundamental characteristic of a distributed system: at a given time, some components of the system might have failed whereas others might be operational. The ability to hide partial failures or recover from them is a crucial metric for measuring the reliability of a distributed system. All reliability schemes that we know make use of some form of failure detection mechanism. Failure detection is a crucial component in transaction processing, replication management, load balancing, and distributed garbage collection, as well as in applications requiring monitoring facilities such as supervision and control systems.

In most distributed systems however, failure detection is left to the application developer. Failures are handled through mechanisms like exceptions and it is up to the programmer to distinguish a physical failure (the crash of a machine) from a logical failure specific to the application's semantics. Some reliable distributed toolkits such as transaction monitors and group communication systems provide some support for failure detection through timeouts. For instance, an object is suspected to be faulty if it does not respond to an invocation after some time. The specific code that handles timeouts is usually mixed up with the code of the distributed protocols in charge of failure-hiding or failure-recovery. For instance, in transaction monitors, the code for timeout management is usually mixed up with the code for distributed transaction manipulation protocols such as atomic commitment. It is very difficult, if not impossible, to adapt the failure detection mechanism to the network topology without modifying the application or the under-

[1] Failure detection abstraction has been implemented and is available at lsewww.epfl.ch/OGS.

lying distributed protocols. The only parameters that are generally left to the developer are the timeout values. These are indeed important parameters that enable the developer/user to trade latency (short timeouts and hence fast reaction to failures) with accuracy (long timeouts and hence more accurate failure detection). The developer cannot, however, tune the failure detection protocol itself according to the network topology. This can be viewed as a serious drawback of existing distributed systems and can reduce their scalability and more generally their applicability in various contexts.

For example, according to the network topology and the communication pattern of the application, the choice between a push (heartbeat) or a pull (are you alive?) monitoring model can have an important impact on the performance of the system. In a push model, every component of the system is supposed to regularly send heartbeat information to the other components: a component is considered faulty if its heartbeats are not received by the other components in time. In a pull model, a component A monitors a component B by sending "are you alive?" messages. If B fails to respond to these messages after some timeout, A considers B to be faulty. Neither the push nor the pull model fits all situations. In a large-scale system, one might use either of those models or even mix them to reduce the number of messages exchanged in the network.

When following an OO modeling style, failure detectors should be considered first-class citizens. That is, failure detection should be viewed as an abstraction, the complexity of which is encapsulated behind well-defined interfaces. The various roles of a failure detection service should all be represented by first-class objects. As a consequence, one can reuse existing failure detection protocols as they are, or define new ones through composition or refinement.

Failure Detection as a First-Class Abstraction

Failure detection can be viewed as a generic service that supports several interaction styles and may be configured in various ways. The interfaces of such a service can be arranged in a hierarchy that provides different views of the service and different interaction paradigms for failure detection. In particular, the hierarchy may include specialized interfaces for push and pull execution styles. A dual monitoring model where the advantages of both styles are combined can simply be obtained by inheriting from both push and pull interfaces.

On one hand, failure detection mechanisms should be separated from other mechanisms in the system to provide for better modularity and extensibility. In fact, even the various roles of failure detection components should be decoupled. The failure detection service can be viewed as a hierarchy of well-defined interfaces. One can reuse existing mechanisms or build new ones through composition or refinement.

On the other hand, the entities being monitored should be abstract objects in the system to eliminate the mismatch between the need for failure detection at the level of application objects and the support provided by some operating systems to detect host failures. One can configure the failure detection service in such a way that the monitored units can range from specific application objects, processes, machines, or even subnets.

Interfaces

Client applications using the service for monitoring remote objects have a limited view of the service, restricted to the three topmost interfaces. These interfaces abstract the flow model used for object monitoring. As a consequence, applications that use the service do not need to care about the interaction paradigms used for monitoring objects. In particular, this makes it possible to mix several monitoring models in the same distributed application with no impact on the functional objects of the application. The three topmost interfaces abstract the roles of the categories of objects involved in a monitoring system:

- Monitors (or failure detectors) are the objects that collect information about component failures.
- Monitorable objects are objects that may be monitored (the failure of which may be detected by the failure detection system).
- Notifiable objects are objects that can be registered by the monitoring service, and that are asynchronously notified about object failures.

Monitorable and notifiable objects are generally application-specific. In other words, the interfaces deriving from monitorable and notifiable are interfaces that the application must support for the service to call back the application. Default implementations of monitorable objects are provided by the service. However, these objects must

be instantiated by the application.

In contrast to the monitorable and notifiable interfaces, monitors are implemented by the service and do not need to be instantiated by the application. More precisely, interfaces deriving from the monitor are service objects, the implementation of which is provided by the service. These interfaces abstract the behavior of monitoring protocols and the way the information about component failures is propagated in the system (the flow policy). There are two basic forms of unidirectional flow, push and pull, plus several variants. These flow policies correspond to simple monitoring protocols.

The Push Model. In the push model, the direction of control flow matches the direction of information flow. With this model, monitorable objects are active. They periodically send heartbeat messages to inform other objects they are still alive. If a monitor does not receive the heartbeat from a monitorable object within specific time bounds, it starts suspecting the object. This method is efficient since only one-way messages are sent within the system, and this may be implemented with hardware multicast facilities if several monitors are monitoring the same objects.

The Pull Model. In the pull model, information flows in the opposite direction of control flow (only when requested by consumers). With this model, monitored objects are passive. Monitors periodically send liveness requests to monitored objects. If a monitored object replies, it means it is alive. This model may be less efficient than the push model since two-way messages are sent to monitored objects, but it is easier to use for the application developer since monitorable objects are passive and do not need to have any time knowledge (they do not have to acknowledge the frequency at which the monitor expects to receive messages).

When following an OO modeling style, failure detectors should be considered first-class citizens.

The Dual Model. Both push and pull interaction models have interesting properties. In the pull model, the monitor parameters (for example, timeouts, which may need dynamic adjustment) need only reside in the monitor and are not distributed in all the monitorable objects. On the other hand, push-style communication between monitor and monitorable objects is more efficient and may reduce the number of messages generated when using hardware multicast facilities (such as IP multicast) if several monitors are listening to the heartbeats.[2] Both models are thus complementary, and the type of interaction to use depends on the application.

[2] Note that heartbeat messages generated by a large number of monitorable objects may also inadvertently flood the network.

Therefore, we introduce a model resulting from the combination of the two models, called the dual model, in which the push and pull models can be used at the same time with the same set of objects. Informally, the dual monitoring protocol works as follows: The protocol is split in two distinct phases. During the first phase, all the monitored objects are assumed to use the push model, and hence send liveness messages (heartbeats). After some delay, the monitors switch to the second phase, in which they assume that all monitored objects that did not send a heartbeat during the first phase use the pull model. In this phase, the monitors send a liveness request to each monitored object and expect a liveness message (similar to the push model) from the latter. If the monitored object does not send this message within specific time bounds, it becomes suspected by the monitor.

The dual model is not a new failure detection protocol per se. It should rather be viewed as a way to mix different styles of monitoring without requiring the monitor to know which model is supported by every single monitorable object. Thus, it provides more flexibility by letting monitorable objects use the best-suited interaction style.

Interactions. There are two types of interactions between the components of the object monitoring service, such as application clients, monitors, notifiables, and monitorable objects as follows:

- Monitor <–> client and monitor <–> notifiable. This interaction allows the application to obtain information about object failures.
- Monitor <–> monitorable. This interaction is performed by the

monitoring service to keep track of the status of monitorable objects.

The basic interaction paradigm of the monitoring service consists of having monitors and monitorable objects communicate with each other using remote method invocations. When using the push execution style, monitorable objects periodically invoke the i_am_alive() operation of the monitors they are registered with in order to advertise the fact that they are alive. When using the pull execution style, monitors periodically invoke the are_you_alive() operation of monitorable objects; this is a one-way operation, and the monitorable objects should react by invoking the i_am_alive() operation of the monitor that originally issued the liveness request. When using the dual execution style, these interfaces allow the marriage of the push and pull models. During the first phase of the dual protocol, the monitor assumes that all monitorable objects use the push execution style, and expects heartbeat messages. During the second phase, the monitor assumes that all monitorable objects from which it did not receive a heartbeat use the pull execution style. Thus, it sends liveness requests to these objects.

The default method used by a monitor to keep track of the status of the components in the system is to periodically check whether they are or are not alive. This information is stored in a local table, and given to clients when asked about the status of a particular object. Liveness information is typically associated with a time-to-live value (which may change on a per-object basis) telling when to invalidate and re-evaluate the suspicion information. Another way to obtain information about the status of monitored objects is to do it on a client's demand (lazy evaluation). With this scheme, the monitorable object is checked on client demand (for example, when the client asks the monitor for the status of an object). This makes the system less reactive since the client has to wait for the liveness request to return before knowing the object's status. This may reduce the number of messages exchanged in the system to perform the actual monitoring.

A client can ask the monitor to start and stop monitoring an object by invoking the start_monitoring() and stop_monitoring() operations, and obtains the status of an object by invoking the is_it_alive() operation. From a monitor's point of view, a monitored object can have one of three states:

- SUSPECTED means the object is suspected by the monitor.
- ALIVE means the object is considered to be alive by the monitor.
- DONT_KNOW means the object is not being monitored.

Although most applications need to invoke the monitor synchronously at specific points during protocol execution, it may sometimes be useful to receive asynchronous notifications when the state of an object changes. In particular, when protocols are implemented using a state machine approach, a suspicion can be seen as an event that may trigger some specific action. In this situation, asynchronous suspicion notifications greatly reduce the complexity of the protocol's implementation. A parameter of the start_monitoring() operation allows the registering of an object with the notifiable interface. The monitor invokes the notify_suspicion() operation for each registered notifiable object when the status of a monitored object changes (if an object becomes suspected, or if an object previously suspected is discovered to be alive). The client may still pass a null reference as a notifiable object if it is not interested in asynchronous notifications.

Scalability Issues

The problem of scalability is a major concern for a monitoring service that has to deal with large systems. A traditional approach to failure detection is to augment each entity participating in a distributed protocol with a local monitor that provides suspicion information. However, this architecture raises efficiency and scalability problems with complex distributed applications, in which a large number of participants are involved. In fact, if each participant monitors the others using point-to-point communication, the complexity of the number of messages is O(n2) for n participants. Wide-area communication is especially costly and increases the latency of the whole system. Thus it is very important to reduce the amount of data exchanged across distant hosts.

The interfaces of a generic failure detection service make it easy to configure the monitoring system in a hierarchy. In a LAN, one or more[3] failure detectors keep track of the state of all local monitorable objects, and transmit status

[3] Redundancy may be required for fault tolerance.

information to remote monitors in other LANs, thus reducing the number of costly inter-LAN requests. Similarly, the developer may choose to install one monitorable object per host, per process, or per thread, depending on the kinds of failures that he or she considers. These configuration choices may be taken at runtime, and do not require modifications in the interfaces of the service.

A monitor may receive liveness information about a specific monitorable object from another monitor rather than directly from the monitorable object. This second-hand information may be obtained by asking other monitors about the status of each individual object or by transmitting complete tables of suspicion information, thus reducing the communication overhead. Note that the latter solution requires an extension to the service's interfaces in order to transmit these tables.

The hierarchical configuration is independent of the model used for monitoring objects (push, pull, or dual model). It permits a better adaptation of monitor parameters (such as timeouts) to the topology of the network or to the distance of monitored objects, and reduces the number of messages exchanged in the system between distant hosts. A monitor located on a LAN can adapt to the network characteristics and provide a specific quality of service. The reduction of network traffic, especially when a lot of monitorable objects and clients are involved, is the main reason for the good scalability of this hierarchical approach.

One can also combine the flexibility of hierarchical dissemination with the robustness of flooding protocols (in which a member diffuses the information to all its neighbors or to all other members) through a gossip protocol where a member forwards new information to randomly chosen members. One can easily build this protocol with a generic OO failure detection architecture, by implementing monitors that occasionally send their suspicion information to other monitors. The interaction between monitors and clients/monitorable objects is not affected.

The reduction of network traffic, especially when a lot of monitorable objects and clients are involved, is the main reason for the good scalability of this hierarchical approach.

Conclusion

Identifying the fundamental abstractions in distributed computing and classifying these abstractions according to some inheritance hierarchy lies at the heart of OO distributed programming. In this column, we have considered one such abstraction, namely failure detection, and we have discussed how it can be modeled using first-class objects. The aim is not to describe a specific failure detection protocol but rather to sketch a modular architecture to compose and customize failure detection protocols according to the topology of the system and the communication pattern of the application.

We do not claim that a failure detection service should be used by all developers. There are indeed many applications where failure detection would just be hidden behind other services that address reliability issues such as group membership or transaction management. However, on one hand the modularity of these services would be increased if failure detection is encapsulated in a separate component. On the other hand, applications such as supervision and control or network management systems directly need to handle failures. Thus it is also important to encapsulate the complexity of failure detection inside first-class (application level) components with well-defined interfaces, namely first-class objects.

Through our failure detection architecture case study, we give a rather concrete example of what OO distributed programming should be. There are many other abstractions in distributed computing for which a similar design can be adopted. C

Pascal Felber (pfelber@us.oracle.com) is a senior engineer at Oracle Corporation, Portland, Ore.
Rachid Guerraoui (Rachid.Guerraoui@epfl.ch) is an assistant professor in the Computer Science Department at the Swiss Federal Institute of Technology, Lausanne (EPFL).
Mohamed E. Fayad (fayad@cse.unl.edu) is an associate professor in the Computer Science Department at the University of Nebraska, Lincoln.