Part One

Object-Oriented Distributed Systems

1

Issues in Distributed Systems

Peter F. Linington
University of Kent, UK

1.1 A Distributed Systems Perspective

No one can doubt the importance of distributed systems these days, but they do pose a particular challenge to the developers of formal methods: they involve, by their nature, a high degree of independence between components and have dynamically changing structures, the evolution of which needs to be captured in any formal specification. They therefore provide an excellent test of the expressive power and flexibility of the range of modern formalisms available for system specification.

One can imagine specification problems as forming a spectrum from purely local problems, through communication between fixed sets of processes, to dynamically evolving distributed systems. A local problem, such as the description of a file storage system or demonstrating the correctness of a lift scheduling algorithm, is characterized by a single context and a monolithic system state and generally involves a fairly simple view of errors. Specification of a protocol to be used for communication between two systems is more complex, because the systems have independent states and are linked by a medium that is subject to errors. Each system can fail or be restarted independently of the others, and so the behaviour to be described is much more complicated. In short, concurrency and interruption of processes have become the norm rather than the exception. These problems are common to a wide range of protocol specification tasks, from simple data transfer protocols to protocols needed for remote object invocation, transaction management or transmission of streaming media, such as video.

Specification of the behaviour of a distributed system is more complex still. In addition to the protocol problems of physical separation and localized errors, there is a wide variety of other problems associated with the evolution of the system's configuration. The description of a distributed system will generally involve fragments of behaviour in which some components create others, and then publish their availability by passing references to yet further objects that can then exploit the new services being offered. This dynamic behaviour implies that a specification must consider not only the inconsistency of the state of the communicating objects, but

it also must deal with inconsistent views of what the set of communicating objects is.

Hand in hand with this increasing emphasis on concurrency and flexible configuration comes an increasing requirement for openness. Once a system is described in terms of a number of distinct, communicating parts the way is open to its realisation using separately implemented components from a variety of sources. The result is an increased emphasis on abstract specification and refinement, because there is more flexibility if the builder of each component is able to choose a style, language and approach without unnecessary systemwide constraints. This is particularly important when considering component reuse and the evolution of legacy systems. It will be easier to incorporate existing components or subsystems if their role in the new system is expressed in high-level terms, without taking account of local details of the implementation of the existing components.

The result of all of these considerations is that thinking about distributed systems leads to a number of shifts in emphasis when compared with centralized systems. In particular there are shifts:

- from central control to autonomy: in a centralized system, there is a single source of control that initiates actions and provides a natural focus for coordination. In a distributed system, each of the components is under independent management and is involved in making independent, unsynchronized decisions. This leads to the potential for much more complicated behaviour, particularly when arbitrary components fail or are reinitialized.
- from global naming to federated local naming: if names are all issued by a single authority, they can be passed to other parts of the system and used anywhere in the system with equal validity and without regard to their origin. If the system contains multiple independent naming authorities, as it must generally do to make the design scalable, there need to be rules for their federation and for ensuring that any name quoted is interpreted correctly in the context in which it was defined, no matter where it is used.
- from shared memory to local encapsulated state: once there is a physical separation between the system components, it becomes impossible for the actions that are to be performed within the system to be specified in terms of arbitrary elements from the system's state. Each component performs actions based on its local state, including actions that communicate with other components, but the behaviour of a component is influenced only by the state of other parts of the system indirectly by receipt of communication events originated by them. Each component maintains a local view of its beliefs about the states of the remaining parts of the system, but this information depends on communication between components for its currency and accuracy.
- from global consistency to weak consistency: since each part of the system is linked to others by communication that may be subject to errors, it becomes likely that the information held by different components will be inconsistent. Invariants and checks on consistency constraints that in a centralized system could be asserted with confidence as true for any correct implementation become ideals that will frequently not be true. The consistency of replicated data, for example, becomes something to strive for, involving a balance with respect to other targets, such as performance and system cost.
- from sequential execution to true concurrency: a single program executing on one proces-

sor will behave in a quite predictable way, with the sequential execution interrupted only at well-defined and controllable points. Thus the language, run-time environment or operating system can provide well-behaved threading and synchronization facilities that can be used to avoid unexpected effects. In a distributed system, the combination of multiple processors progressing independently at different rates and communication with variable delays means that practically any combination of event timings can occur and must be taken into account in describing the behaviour that the system is expected to have. The necessary synchronization of components must be achieved by explicit communication between them.

- from vulnerability to fault tolerance: one of the major benefits of distribution is that errors affect individual components, not the system as a whole. It therefore becomes possible to make the complete system much more robust, because the remaining parts of the system can cooperate to mask the consequences of a failed component. However, this resilience comes at a cost, because the rules and procedures that make the system fault tolerant need to be defined within the system's specification.
- from fixed location to process migration: in a centralized system, location is not an issue. All of the elements of the system are in the same place, and so they can always communicate. Once there is distribution, each part of the system needs to know how to find any other component or service in order to communicate with it. This may be a matter of finding a component when it is first needed, or it may be a matter of tracking a component as it moves, either to another hardware platform or as a result of the supporting hardware itself moving in a wireless or nomadic computing environment.
- from fixed configuration to continuous evolution: once a system is made up of separate components, it becomes possible to upgrade or replace these components while the system is running. Any sufficiently large-scale system is unlikely ever to be initialised as a whole at one time, and the classical waterfall approach to system development ceases to be even a theoretical possibility; there are no more new systems, just incremental changes or extensions to some existing system. Even a new application will generally have dependencies on existing networked services and legacy data.
- from early binding to late binding: in building a monolithic system, it is possible to identify during construction how all of the parts interact, and to perform a considerable number of checks on the consistency of the design, by, for example, applying strong type checking to the linkages made. In a distributed system, the components to be linked may not be identified until after the system has begun operation. There is much more flexibility in configuring the system because of this late binding of components, but it is at the cost of the need to perform some of the necessary checks for correctness while the system is running, and to cope with any failures at that stage. One of the objectives to aim for in providing support for system development is to maximize both the flexibility and the type safety of component binding.
- from a single view of time to multiple clocks: it is a natural consequence of the autonomy of the different hardware components that make up a distributed system that the way in which time is measured by the different components will vary. Clocks will drift, and it will be practically impossible for different parts of the system to initiate actions so that they are guaranteed to be simultaneous. Whatever level of synchronization between clocks is necessary must be constructed by a suitable pattern of communication between the components.

- from homogeneity to heterogeneity: once a system is constructed from independent components, the emphasis shifts from how the components are to be constructed to the external properties that each component must display in order to fulfil its role in the system. This approach makes it possible to construct different components using different languages, platforms or design methodologies. These differences are unimportant as long as the required externally observable behaviour of each component is maintained. The specification of the external interactions becomes paramount.

1.2 Current Issues in Distributed Systems

In addition to these general changes of perspective, there are a number of particular problems or areas of concern that characterize work on distributed systems. These are recurrent issues in research, and solutions to them are likely to be prominent on a list of requirements drawn up by any distributed systems developer.

1.2.1 Federation

Large distributed systems are generally long-lived, and they often span organizational boundaries. They need to evolve to meet changing requirements over a long period of time, and they are typically supported by loosely coordinated management and maintenance teams with divided responsibilities and objectives. Thus any large system is likely to fall into a number of distinct management domains, and any coordinating authority will operate on the basis of a fairly high-level view that is imperfectly communicated to the individual domains. In other words, large distributed systems generally have a federated structure and so their development must pay particular attention to the problems of federation.

What, then, are the characteristics of federated systems? First, there is no single controlling component. Instead, the members of the federation communicate in a number of peer-to-peer dialogues, and, in consequence, federated structures are well-suited to the support of large-scale activities. Second, the constraints placed on members say as little as possible about their internal organization, reducing the cost of federation between systems of different designs and focusing attention on the points at which information crosses system boundaries. This is because incompatibilities often can be solved by translation at these boundaries. Third, the behaviour of a federation is defined in the knowledge that its members will retain a high degree of autonomy and may choose to join or leave the federation at any time, without protracted negotiation processes.

1.2.2 Legacy Systems

One distributed systems problem of great practical importance is the assimilation of legacy systems [BS95]. This can be seen as a special case of federation, since the problem generally arises because there is no practical possibility of making significant

changes to the internal structure of the legacy system. The most common approach to solving the legacy problem is to encapsulate the system concerned, identifying the services that it provides and providing a wrapper that offers them to other components via standard interfaces.

The analysis of what a legacy system does, and the formulation of interface definitions that capture only that service, without commitment to a host of obscuring implementation detail, is not an easy task. It needs techniques for abstract specification and for the creation of suitable transforming gateways that provide the translation between the different ways in which the old and new systems refine the overarching description. This is not just a matter of translating message syntax; it also needs to take into account things like the transactional implications of sequences of messages and possible differences between the failure models of the old and new systems.

1.2.3 Distributed Systems Management

Telecommunications networks have always had a need for large-scale distributed management systems. However, these used to be thought of as a largely separate activity, identified architecturally as a separate management plane and supported by special-purpose notations and protocols. As the networks being managed have become less focused on special services, such as voice, and increasingly concerned with the general problem of data transmission, the rationale for separate management facilities has been eroded. There is a move in the newer architectures toward seeing management as just part of a more general distributed systems problem.

However, there is still a tension between the system management and the application builder's view of design. The application builder tries to hide details of infrastructure and internal configuration of components, because exposing such details adds to the constraints under which evolution must be performed, leading to brittle systems. Thus a service is accessed by reference to some interface without exposing the configuration of objects supporting the service or necessarily making visible any other interfaces that those objects support. This leaves sufficient flexibility to reengineer the system as part of an evolution plan, changing the way in which a service is provided and its functions packaged in a particular implementation.

In system management, on the other hand, there is a need for the manager to be able to start from a particular interface, which might, perhaps, have been behaving incorrectly, and to navigate through the configuration of objects supporting the service. This leads to a requirement for a different view of the system, in which one can go from knowledge of an interface to at least knowledge of a management interface responsible for the resources supporting it.

Techniques have recently been introduced that satisfy both requirements, by allowing controlled visibility and manipulation of the properties of the implementation supporting a given service. Component models (see, for example, Chapter 18) provide one approach; another is based on the idea of reflective programming (see

Chapter 14), which allows properties of the implementation to be explored in a uniform and systematic way and makes configuration control much more flexible.

1.2.4 Security

Security is particularly important in distributed systems because physical restrictions on access are no longer an option. Extreme measures, like putting the data processing system inside the bank vault, are of no help if the aim is to support global e-commerce.

There is an immediate apparent conflict between the requirements of security and openness (see Chapter 10, which discusses, for example, security barriers and firewalls). If the aim is to avoid unnecessary barriers to communication between systems by ensuring that there is always a basic level of compatibility, the mechanisms for authentication and privacy become more important, and the specification of security requirements needs to be integrated with other aspects of system design.

Security places some special demands on techniques for system description. In most situations, the objective is to show that the system does something – that it performs some particular behaviour without deadlock, for example. In the security field, however, the requirement is often to show that broad classes of behaviour cannot happen. Thus it is important to show that a design is fully encapsulated, allowing communication only at declared interfaces – that it does not leak. This is a much more difficult problem than showing that it is deadlock-free or refines a given service specification.

1.2.5 Multimedia and Real-Time Systems

Multimedia and real-time requirements (see Chapters 16 and 17) are not inherently related to distribution, since they also occur in localized systems. However, like security, multimedia support is likely to be expected of any general-purpose modern system, and the real-time deadlines that are involved are aggravated by distribution. Continuous media in particular presents new problems both to system implementors and to the designers of description techniques [BS97]. This is because, just as they are continuous, these media require a steady and predictable sequence of media samples, implying a steady succession of cumulative hard timing targets. The resource management at all points along the communications path traversed by the continuous media needs to be coordinated to give the necessary guarantees.

The exact way in which this resource management is carried out will vary from component to component, as a result of different constraints and implementation strategies. This is particularly true where media are multicast to different receivers, which themselves have disparate infrastructure capabilities. An abstract description of a multimedia application should therefore express multimedia traffic in terms of unstructured flows, not of the sequences of samples that represent the medium, to avoid complicating the description with details that are subject to local variation.

1.2.6 The Development Process

To keep the system development costs within reasonable bounds, there need to be development tools and techniques that help to manage the complexity of large systems and which encourage reuse of both previous implementation fragments and services and of their specifications. This implies a need for both the system and its specification to be modular, and for it to be possible to manipulate these modules, combining them in ways not foreseen when they were produced.

The implication for the designer of specification languages is that there need to be good facilities for producing and combining families of specifications, either considering different abstractions or different subdomains within the system. The viewpoints discussed in the next chapter are an example of such a specification framework (see also Chapter 20). Another part of the problem can be solved by concentrating on components that can be composed while retaining interesting aspects of their behaviour in the composition (see Chapter 18).

Support of large teams in which different members may be concentrating on different aspects and using different tools and notations implies a need for mechanisms that support sharing of component specifications and management of their consistency. This is one of the motivations for the creation of integrated development environments, but these often lack flexibility. A more loosely coupled federated approach is attractive, in which emphasis is on shared access to the various specifications via a distributed repository. This leads to tool families centred around shared repositories, such as a type repository or a meta-object facility. This is one of the motivations for subtyping techniques, which are discussed in Part 4.

1.2.7 Potentially Distributed Systems

Although the change of perspective from local to distributed systems is a necessary result of their division into communicating components, many of the approaches taken to support distribution also solve other problems of system structure. Even localized systems benefit from an architecture in which there is a strong encapsulation of processes and protection between different security and management domains. It is no accident that modern operating systems tend to be structured into a minimal kernel and a set of objects providing supporting services. Neither is it surprising that such systems lend themselves well to distribution when the need arises. One of the recognizable diseases of old age in an operating system is the addition of performance-enhancing features that break encapsulation and increase the cost of further evolution. This 'hardening of the linkages' reduces flexibility and adds to maintenance costs.

All systems, then, are potentially distributed. There are significant benefits from architectures and design styles in which there is a well-defined separation into components, with clearly identified points of interaction between them. It is the local

systems that are the special case, in which the need for actual distribution has not yet been identified.

1.3 Object-Oriented Design

The key features of an object-oriented system are traditionally taken to be the identification of objects, with classification and inheritance as tools to support their description [Weg86]. For a more recent general review of object orientation, see [BGHS91]. The emphasis on encapsulated objects comes directly from regarding the system as made up of distinct things, which can most simply be considered as objects. This is a natural requirement when considering distributed systems because the elements of a distributed system and the resources that they depend on are individually localized, and the study of a distributed system is, in large part, the study of the interactions of the objects that make it up. It can be argued that viewing things as objects underlies our ability to use language to refer to the real world. Objects emerge from the need to simplify and condense the welter of observations that we make, and we carry this process further by identifying common attributes of the objects perceived [Qui74].

Classification is also fundamental to the way that humans think. Our first step in simplifying or generalizing problems is to identify classes of objects with common characteristics or properties, either in terms of common attributes or common patterns of behaviour. There are long-standing debates as to the relative importance of feature identification or prototype identification (see [Lak86] for one clearly argued view), but there is universal agreement as to the key role of the process of classification in design and specification. Classes as they appear in object-oriented computer languages provide templates for creating objects. They include both the state the object will maintain and the methods that represent its behaviour. The class of an object is thus more prescriptive than the type, which tends to concentrate on the observable properties of the object. This distinction is particularly important in distributed systems, where systems are not constructed as a single comprehensive exercise, but frequently need to incorporate a variety of preexisting components from different sources.

Classifications are rarely flat; making them so involves too much duplication of material, and so something that brings out common features to form a structured classification system is normally adopted. The result is either hierarchical, forming a tree in which classes become more specific when descending from some general root, or based on a more general directed graph, allowing classes to take on the properties of more than one abstract precursor. In either case, the specification of the classification system can be managed by inheritance, so that properties stated once for some node in the structure are implied to hold for all of its descendents. The pure hierarchy relies on single inheritance, and the more general structures exploit multiple inheritance. A discussion of inheritance will appear in Part 4, while a detailed theory of inheritance is documented in Chapter 15.

Inheritance is not just a matter of avoiding duplication; it has positive benefits in making clear where common features are intended rather than accidental. Good use of inheritance avoids most of the need for simultaneous updating and the manual preservation of consistency that would occur if information was duplicated in related definitions, and so the use of inheritance makes specifications easier to maintain.

1.3.1 Inheritance and Distribution

In programming languages, the classification system that is used is an integral part of the language, and there is, increasingly, a tendency to exploit this integration by allowing reflection, so that the running program can take advantage of knowledge of its own structure. Inheritance is therefore seen as fundamental. In specifying a distributed system, which may incorporate existing components in a more open way, as indicated above, things are less clearcut. The specification is not necessarily a template, it may not be in an open form and it may even postdate the component specified. It is for this reason that distributed systems' frameworks are often object-based rather than object-oriented. They depend on the ideas of object and class (often formulated with particular emphasis on the encapsulation boundary by focusing the specification on interfaces), but they are not committed to the strong linkage between system and specification that is needed to exploit inheritance.

This difference might be epitomized by contrasting the Smalltalk programmer with the Java RMI programmer. The first assumes that code written to modify the object representing a class will have an effect that is immediately visible as a change in behaviour of all objects so far created from that class. The second knows that remote invocation can result in a copy of some class being transferred to a remote virtual machine, where it can remain in its original form, even if the original class is updated and replaced locally.

There is still, of course, a need to describe the capabilities of system components and to negotiate a match between requirements and these capabilities. However, a more descriptive approach, based on subtyping and type matching rules (see Chapter 11), is generally used to support open binding mechanisms. Where inheritance is used, it is applied at a specific epoch associated with the binding process.

1.3.2 Problem-Domain Objects and Implementation Objects

Object-oriented specifications come in a wide variety of styles and levels of abstraction. In applying object-oriented techniques to the analysis of system requirements, the objects are generally models that stand for some part of the problem domain and translate into representations of the problem domain that are manipulated by the system that is eventually produced.

When discussing the structure of distribution platforms and the way in which distribution mechanisms are organized, the objects are more directly tied to system components and are often used to encapsulate particular system resources. Object