1

Introduction

Most modern computing systems are distributed in nature. A *distributed computing system* consists of multiple autonomous processors, often heterogeneous, that do not share primary memory, but cooperate by sending messages over a communication network in order to achieve a common goal.

In the late 1980s *middleware platforms* were introduced to support distributed computing systems. Middleware platforms run on heterogeneous operating systems and communication systems, providing a homogeneous abstract view of the entire distributed system. Most middleware systems like CORBA [OMG02b] or Java RMI [Sun99b] are *invocation-based* and thus follow a *request/reply* paradigm: a client requests a particular service from a server by either sending a request message or performing a remote method invocation (RMI) and then receives a reply in return.

The size and complexity of distributed systems have been increasing inexorably. With the advent of the Internet, it became possible to build *large-scale distributed systems* because of the existence of a global, *packet-based* communication infrastructure. This increase in system scale results in systems with millions of nodes that need to communicate and cooperate in order to achieve the common goal.

Traditional middleware work well in a network with a moderate number of clients and servers, although they are inadequate for large-scale systems. This is mainly because the request/reply paradigm only supports *one-to-one* communication where a single client interacts with a single server. In contrast, large-scale systems benefit from *many-to-many* communication since the client does not have to decide on the best communication partner. Another problem is the *tight-coupling* of request/reply middleware: the method invocation is synchronous forcing the client and server to couple at one particular point in time. Such a behavior is clearly not desirable on the Internet because of the large number of potential

communication partners and the dynamic nature of the system with new clients joining and servers failing.

Furthermore, the proliferation of execution platforms with technical features very different and the democratization of networks, confront to execution contexts more variables:

- *variations in the space*, the large diversity of platforms covers a wide spectrum in terms of available resources;

- *variations in the time*, the execution context of a system evolves during its execution due to the availability of physical and logic resources, mobility, etc.;

- *application requirements*, a system can be used in situations which require different operating modes, and by users with different expertise levels and with specific needs.

This situation rends the system development more complex, because it is difficult to know at the development time, the precise conditions in which the systems will be used. Even when that is possible, these conditions will be induced to evolve due to unforeseeable reasons during the system time life.

These increasingly diverse and dynamic contexts in which current systems are run impose to *adapt* and to become more *autonomous*. The new systems must not only be able to be adapted, but also to be able to adapt itself in an autonomous way to the many and various execution contexts to which they will be confronted, and to the evolutions – increasingly dynamic – of those.

An *adaptation* is a modification of a system in response to a change in its context. The resulting system is better suited to perform its function in the new context. A system is *adaptable* if it can be adapted by an external entity and a system is *adaptive* if it adapts itself automatically and in an autonomous way.

Event-based communication has become a new paradigm for building large-scale distributed systems built out of heterogeneous components, using terabytes of structured, semi-structured or non-structured data and accessed by thousands of users. It provides *asynchronous*, *loosely-coupled* and many-*to-many* communication, being scalable, and providing a simple application programming model.

In *event-based systems*, events are the basic communication element. An *event* can be seen as a notification that something of interest has occurred within the system. System components act as either *event consumers* that express their interest in receiving certain events in the form of an *event subscription*, or *event producers* that publish new events which will be delivered to all interested consumers. The service that connects producers and consumers is named *event service*. Therefore, an event service notifies to consumers about produced events of interest.

Large-scale distributed systems call for event services that integrate event-information from different sources under various or changing applications. Integration requires the notion of *composite events* that are combinations of simpler events.

1.1 Event services

Event services implement (i) an *event model*, which describes the events that can occur in a system, and (ii) an *event management model*, which describes the event management functionalities.

Event management is based on the *producer/consumer* approach. Event-based system components act as either event producers that publish new events, or event consumers that subscribe to events by providing a specification of events of interest to them. Then, consumers are notified of any event generated by a producer, which matches their interest. The event management starts when the event is detected, and ends when it is notified. Figure 1.1 shows the three phases of the event management.

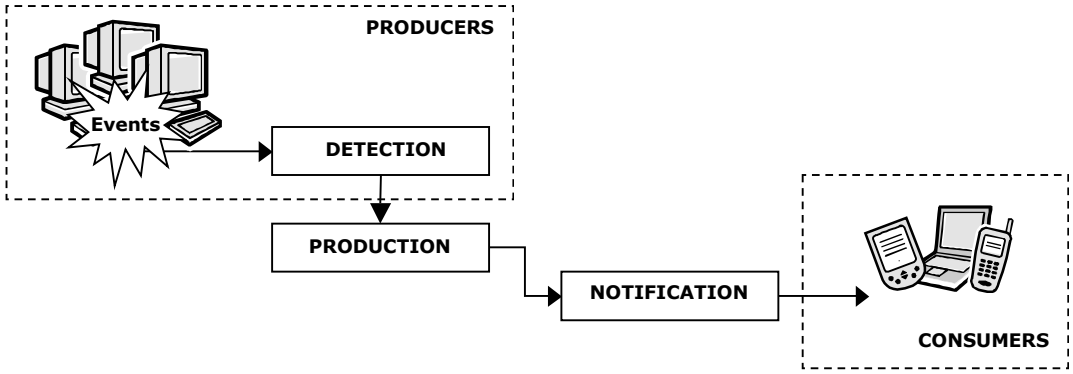

Figure 1.1 Event management

Detection is the process by which an event is recognized and associated to an occurrence instant. Events can be observed by a process external to the producer or the producer can explicitly signal its occurrence.

Production corresponds to the insertion process of a set of detected events in an *event history*. Events are ordered with respect to other events and considering theirs occurrence instants. Once ordered, events can be used to build composite events combining the produced events with composition operators such as disjunction, conjunction or sequence. Composite events are stamped and ordered too with respect to a production instant which must be calculated considering the semantics of the composition operators.

Notification is the process that notifies events to consumers. The event notification can be done at specific instants with respect to their production instants and considering temporal constraints. Events can be filtered too before being notified.

Event-based monitoring offers an approach for observing how distributed computing systems behave. Specified primitive and composite events are monitored at arbitrary sites of a distributed system. Detected events cause reactions, for example, notifying users, taking some emergency measure or adaptive functions.

Hence, *event-based infrastructures* are well-adapted to programming not only large-scale distributed systems, but also adaptive systems that require monitoring its functionality.

1.2 Problem statement

Current event services implementations either do not support integration or they are too rigid and do not provide sufficient solutions. Therefore, it is necessary a service that flexibly supports various and changing applications requirements and enables efficient integration of information from various producers. It implies the event model extension to more flexible and general models.

The extension of event models towards more flexible and general models imposes a reflection in the semantics and the processing that is wanted to give to the events. This implies the uncoupling of the event modeling and the application specification and, the design of methods that allow the event type definition independently of the aspects that concern their management (detection, production, notification). Therefore, the aspects concerning the management must be characterized too.

Most of the event models offer the traditional operators of disjunction, conjunction and sequence. The operators allow expressing the composition of events (primitive or composite) to denote new event types. Event composition varies according to the different application requirements, in other words, the semantics of composition operators and in the different production policies.

Furthermore, each type of application needs to have different events types expressed like regular expressions or like objects (messages). This verifies the need to have mechanisms that allow to the personalization of (i) event types; (ii) the composition operators and their associate semantics; and (iii) the composition algorithm to produce composite events.

1.3 Objective and approach

The objective of this work is to specify an adaptable and extensible event management and composition infrastructure for building adaptive systems. *Framework* infrastructures provide more elaborated forms of adaptability and extensibility. Therefore, the proposed infrastructure is an event-based framework which is suitable for designing and implementing personalized event management and composition systems.

Frameworks separate commonalities from variability in an application domain. They are implemented as skeletal groups of software modules that can be adapted for building domain-specific applications. They provide reuse in the form of pre-programmed logic that can be customized to specific needs in that application domain. Therefore, frameworks differ from standard applications since they need to be instantiated.

Current software frameworks are usually implemented in object-oriented languages, nevertheless, recent research on aspect-oriented frameworks is in progress. From an object-oriented point of view, frameworks are described in terms of concrete and abstract classes and a set of flexible points or *hotspots* that together collaborate for the overall software implementation. Hotspots may have different implementation for each framework instance, and are left incomplete until instantiation time. From an object-oriented point of view,

hotspots are usually implemented using abstract classes, template methods and interfaces. The instantiation process consists of completing the framework hotspots in order to create a fully functional application. Some hotspots may require compile-time instantiation (the hotspot has to be configured before the application is running) while others may require run-time instantiation (the missing information is completed only during run-time).

The use of frameworks can reduce the cost of developing an application by an order of magnitude since it promotes the reuse of both design and code. Moreover, they have been adopted in a large set of applications and, for being built upon existing object-oriented programming languages and techniques, they can rely on existing extensibility and polymorphism features from these languages. Because frameworks promote reuse, they can be used to consolidate the domain knowledge acquired during earlier projects so it can be reused in future projects to realize the application goal. Finally, frameworks also hide internal application details, and provide a general domain model, allowing their users to concentrate in customizing the hotspots for their particular needs, instead of being required to understand all the aspects of the program.

A disadvantage of the frameworks is the high initial development cost, which requires a thorough understanding of the domain being automated and the requirements. Hence, the design and implementation of software frameworks is not a trivial task, a balance between the number of features provided by the framework and the hotpots must be reached. An ideal framework includes all common features of a domain and leaves all variability to be implemented as extensions. If the framework includes too many features, it can become complex and less flexible; whereas, if it omits common functionality, its generality gets compromised and different applications will need to implement the missing functionality, which may result in code replication.

The main requirements of a framework are:

Scalability

A standard requirement for any information dissemination system is *scalability*. Scalability refers not only to the numbers of producers and consumers, and the numbers of notifications and subscriptions, but also to the need to discard many of the assumptions made for local-area networks, such as low latency, abundant bandwidth, homogeneous platforms, continuous and reliable connectivity, and centralized control. The publish/subscribe communication model is intrinsically scalable because publishers and subscribers are only loosely-coupled, and the implementation of the event-based framework must take advantage of this.

Expressiveness

The *expressiveness* determines how fine-grained the data model that is offered to producers and consumers of events is. The level of expressiveness influences the algorithms used to route and deliver notifications, and the extent to which those algorithms can be optimized. High expressiveness, as in a content-based system, is desirable from a consumer point of view. Nevertheless, as the expressiveness of the data model increases, so does the complexity of the algorithms. Therefore, the expressiveness of the data model influences the

scalability of the implementation, and hence scalability and expressiveness are two conflicting goals that must be traded off.

Manageability

An event-based framework in itself is a complex distributed system with many components. Easy manageability of such a system is an important requirement, especially because any large-scale system may substantially evolve over its lifetime. For instance, when new system components are added to increase performance or availability, the event-based framework has to adapt without significant amounts of human intervention. The management effort can be reduced by making components as autonomous as possible. Self-adapting systems can relieve the manager from many decisions and thus facilitate the task of system management.

Reusability

The reusability of a framework is an important factor. The stable interfaces provided by frameworks enhance reusability by defining generic components that can be reapplied to create new applications. Framework reusability leverages the domain knowledge and prior effort of experienced developers in order to avoid re-creating and re-validating common solutions to recurring application requirements and software design challenges. Reuse of framework components can yield substantial improvements in programmer productivity, as well as enhance the quality, performance, reliability and interoperability of software.

Extensibility

Framework *extensibility* is essential to ensure timely customization of new application services and features. The event-based framework should support the deployment of new systems. A framework enhances extensibility by providing explicit hook methods that allow applications to extend its stable interfaces. Hook methods systematically decouple the stable interfaces and behaviors of an application domain from the variations required by instantiations of an application in a particular context. Extensibility in an event-based framework can be achieved with a modular design. The framework is partitioned into core components that are always needed and a set of domain-specific extensions that provide optional services.

Interoperability

Due the heterogeneity of large-scale systems, *interoperability* between different forms of systems becomes an important requirement. To improve interoperability, an event-based framework should be built on open standards that are platform- and language-independent. For example, XML as a messaging format simplifies the translation between different formats. The API exported by an event-based framework should include bindings to several programming languages. This allows the distributed application programmer to choose the most convenient language for the implementation of a client.

Reflection

Reflection was first introduced as a programming language concept for languages that can reason about and act upon themselves. Features found in reflective systems should therefore be part of any modern framework, enabling it to inspect and modify its own components and their behavior. This helps the framework operate in dynamic environments, where the application requirements and the underlying network properties, such as resource availability and link connectivity, are constantly changing during the lifetime of the system. Its architecture is usually component-based so that parts of the framework can be replaced and reconfigured at runtime. Hence, reflective features in a framework for large-scale systems are essential to cope with changes in the environment.

Therefore, any novel framework design, such as an event-based framework, should be component-based and harness reflective techniques.

1.4 Document organization

The remainder of this document is organized as follows: chapter 2 presents the event concepts that will enable the understanding of this document and to explore the way in which an event can be modeled. Chapter 3 presents and describes event-based middleware services. Chapter 4 presents the proposed framework for event management and composition in order to build adaptive systems. It is defined as a middleware framework which is suitable for designing and implementing personalized event-based systems. The implementation of a personalized event-based system is presented in chapter 5. Finally, the conclusions and perspectives of this work are presented in chapter 6.