Generic Support for Distributed Applications

Existing and emerging applications require timely response to asynchronous events and secure interoperability of independent services in large-scale, widely distributed systems. To meet these needs, Cambridge University researchers developed middleware extensions that provide a flexible, scalable approach to distributed-application development.

Jean Bacon
Ken Moody
John Bates
Richard Hayton
Chaoying Ma
Andrew McNeil
Oliver Seidel
Mark Spiteri
University of Cambridge

In the late 1980s, software designers introduced middleware platforms to support distributed computing systems. Since then, the rapid evolution of technology has caused an explosion of distributed-processing requirements. Application developers now routinely expect to support multimedia systems and mobile users and computers. Timely response to asynchronous events is crucial to such applications, but current platforms do not adequately meet this need.

Another need of existing and emerging applications is the secure interoperability of independent services in large-scale, widely distributed systems. Information systems serving organizations such as universities, hospitals, and government agencies require cross-domain interaction. These systems must handle large numbers of users and must be easy to use, internally efficient, secure, reliable, and robust. Moreover, independently developed services must work together with minimal extensions to their basic software.

At the University of Cambridge, in the Computer Laboratory's Opera Research Group, we have developed middleware extensions to address these problems: the Cambridge Event Architecture for asynchronous operation and Oasis, an open architecture for secure interoperating services. Our work is wide in scope, with applicability to many multiservice information systems.

ESTABLISHED SUPPORT

Distributed applications must run on a variety of hardware and operating systems. Middleware makes this possible. Middleware is a layer of software that runs above heterogeneous operating systems and communications systems, providing a uniform interface to distributed applications. Current middleware platforms for distributed applications are based on the following software model and architecture.

Distributed-software model

Whatever the physical architecture of a distributed system, we must establish a software model that defines the entities that comprise the distributed system, how they interoperate, and how to specify their behavior. The object model, which underpins all recent middleware platforms, is a realistic basis for software design, including persistent-data management. An object model gets its broad applicability from its separation of interface and implementation. Using a standard for specifying object interfaces, application developers can compose systems from distributed, independently developed, heterogeneous objects.

For example, the Object Management Group (OMG) and the Object Data Management Group (ODMG) provide widely accepted, compatible standards for both transmitted and stored data objects.[1,2] The OMG Common Object Request Broker Architecture (CORBA) defines objects with a standard interface definition language (IDL) that specifies object names, method names, and their arguments in terms of base types and constructors. The CORBA object model does not provide a sufficiently rich set of constructors for database management, so the ODMG has extended it to support object data. The ODMG standard includes the data definition language ODL, which extends OMG/IDL, and the strongly typed

0018-9162/00/$10.00 © 2000 IEEE

functional query language OQL. The CORBA and ODMG standards support many programming-language bindings, allowing software components written in different programming languages to interoperate. Douglas Barry and Torsten Stanienda[3] show how an ODMG binding for Java supports persistent Java objects.

Distributed-software architecture

The software architecture defines how the communicating entities in the distributed-software model are named, located, and protected, as follows:

Principal. Principals are entities that can initiate actions and to which access rights can be assigned. The general term includes an authenticated, logged-in user, a process acting on behalf of such a user, and a process executing a named program.

Service. A service is offered by one or more servers. A server is an object that carries out operations in response to requests from principals (its clients). A service can function as an object manager for objects of a specified type.

Naming and name-to-location binding. A system can provide a core middleware function—name-to-location binding—transparently as part of platform services or as a specific system service. A service must name the principals that can use it. Each domain in a distributed system is likely to register its users and allocate them identifiers. In many systems, logged-in users, named by user IDs, are the only principals. More generally, services can define names specific to the roles of their clients.

System services. Some services have generic roles in a system. For example, a name server (sometimes called an interface trader) can publish a service's interface specification. A system can provide an authentication service based on encrypted passwords associated with persistent principals. Specific services can be associated with applications such as banking.

Access control and authentication. Access control is important to individual services. Each service must specify the principals that can use it and how they can use it. This assumes that the service can identify named principals and associate them securely with persistent agents. Communications between principals and services must be securely signed. If the service is an object manager, it must ensure protected access to objects and provide a means of expressing and enforcing access policy.

Independent administrative domains. Distributed systems are partitioned into domains that reflect administrative and management responsibility. Domains also provide the means for managing scale. The Domain Name System, which defines nested management domains, manages the communications services that support distributed-systems platforms. A distributed application such as an automated room-booking system operates only for a single university department—that is, a single domain. More often, an application operates across multiple domains—for example, the Web's information system.

Heterogeneity and open interoperability. Because the world will not restrict itself to using a single hardware platform, operating system, or programming language, middleware platforms are designed to cope with heterogeneity. It is unrealistic to assume that a single middleware platform will dominate the market, so the major middleware players are committed, in theory, to supporting interoperability. However, extensions to proprietary systems such as DCOM, ActiveX, Java, and JavaBeans are subject to change without notice, making their interoperability problematic.

To support emerging applications, today's middleware platforms require extension in two areas: asynchronous operation and secure interoperability.

SUPPORT FOR EMERGING APPLICATIONS

To support emerging applications, today's middleware platforms require extension in two areas: asynchronous operation and secure interoperability.

Asynchronous operation: events

The model for existing platforms is synchronous method invocation: An object remains passive until a principal performs an operation on it. This is inadequate for many applications in which asynchronously occurring events should trigger an immediate system response. For example, a credit card cancellation operation by a banking service invalidates the record associated with the card at that service and adds the card to a list of "hot" (invalid) cards. If a thief is using the card, the banking service must invalidate it immediately and notify all affected services automatically.

If synchronous method invocation is all that's available, the options are periodic polling and synchronous callback. Frequent polling to learn whether events have occurred overloads communications. Infrequent polling delays the response to individual events so that users perceive the application as sluggish and inadequate, or even open to hostile attack—by observing the polling rate, an enemy might identify a time slot in which to do damage. Event notification by synchronous callback from source to clients carries the risk of delaying the caller if any of the called clients fails or is slow to respond.

Asynchronous operation supports the following application types:

- *Group interaction.* Suppose an online-conference participant draws on a shared whiteboard object. The other conference members should be able to see the drawing immediately.
- *Multimedia support.* The control of multimedia

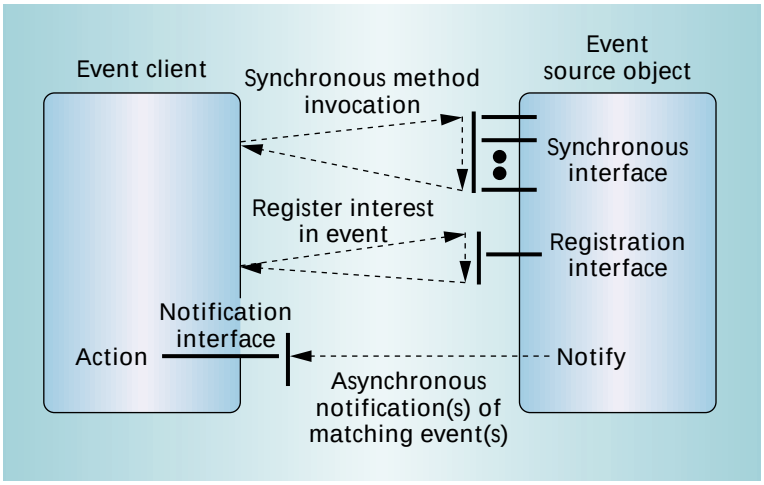

Figure 1. A publish-register-notify event architecture. The object publishes its interface, specified in IDL in the usual way, including the events it will notify. A client invokes the object synchronously, as usual, and can also register interest in events indicating parameters or wild cards. The object asynchronously notifies the client of events that match the registration template, subject to access restrictions.

objects is synchronous, by means of methods such as *pause*, *restart*, and *fast-forward*. At the application level, asynchronous events can trigger these invocations through rules. When a user clicks on the image of a person on screen as a film runs, the system invokes the *pause* method on the video object and shows details of the person in a new window.

- *Mobility.* Mobile users and computers detach from networked systems and reattach later at other locations. An office system tracks users by means of an active badge or a similar infrastructure. Another system locates tourists as they move about a radio-networked city. The detection of a mobile user or computer is an event that may require a system response.
- *Alarms and exceptions.* A security violation, an access control check failure, a system overload, a patient's emergency medical condition, a network component failure, a stock market crash, an imminent nuclear reactor meltdown—all are events requiring rapid system response.
- *Management.* When faults, heavy loading, or resource allocation problems occur, network and system management servers must receive instant notification so that they can take immediate action.

Security

An access control scheme based on an access matrix, in which each entry specifies a given user's rights to invoke a particular object's methods, is inadequate. It neither scales for large numbers of objects and users nor captures the relationships of services. For example, any student known to the student registration service can be a reader of the university library service, which specifies access permissions for broad categories of documents. Independently developed and managed services must cooperate, but mutual trust must be limited. An access control architecture should impose as little as possible on each service while allowing secure interaction between services. Services running in different management domains need mechanisms that allow them to negotiate and interoperate. A student registration service, the local campus library service, and a remote library service at a research institute may all be involved in negotiating access rights for broad categories of users.

Our approach

The Cambridge Event Architecture (CEA) supports asynchronous operation by means of events, event classes, and event occurrences as object instances. For example, an active badge event of type *seen* has the data attributes *person* and *location* and methods that allow these attributes to be read. CEA follows a publish-register-notify paradigm with event object classes and source-side filtering based on parameter templates. It incorporates standard platform technology: IDL for publishing events and automatic stub generation for event notification. Asynchronous notification allows a system to respond immediately to events—for example, the detection of a mobile user or the withdrawal of an individual's access rights. Developers can also use asynchronous event notification to compose applications from independently developed components.

Oasis provides the secure interoperability of independently developed services. For scalability, an Oasis server can name its many users in terms of roles with associated access rights. Entry to a role is restricted to users who can prove they belong to other roles of this or other services. Although Oasis can run above a synchronous platform, asynchronous operation allows immediate response to events.

The term "event" describes asynchronous occurrences of interest within applications—for example, within programs to manage user interfaces and sensor systems. Toolkits that help manage event-based systems have been developed on a system-specific basis. In JavaBeans, for example, a listener can register interest with an event raiser as in CEA. However, JavaBeans is a proprietary, single-language system, and, when a distributed implementation is required, notification is built above synchronous Java RMI (remote method invocation). In contrast, CEA is an asynchronous, language-independent, interapplication platform in an open distributed world. We have integrated Oasis access control with CEA.

CEA

Our work has shown how any widely used platform, such as CORBA, Java RMI, or DCOM, can be extended very simply to support asynchronous operation. The extension enables any object to publish in

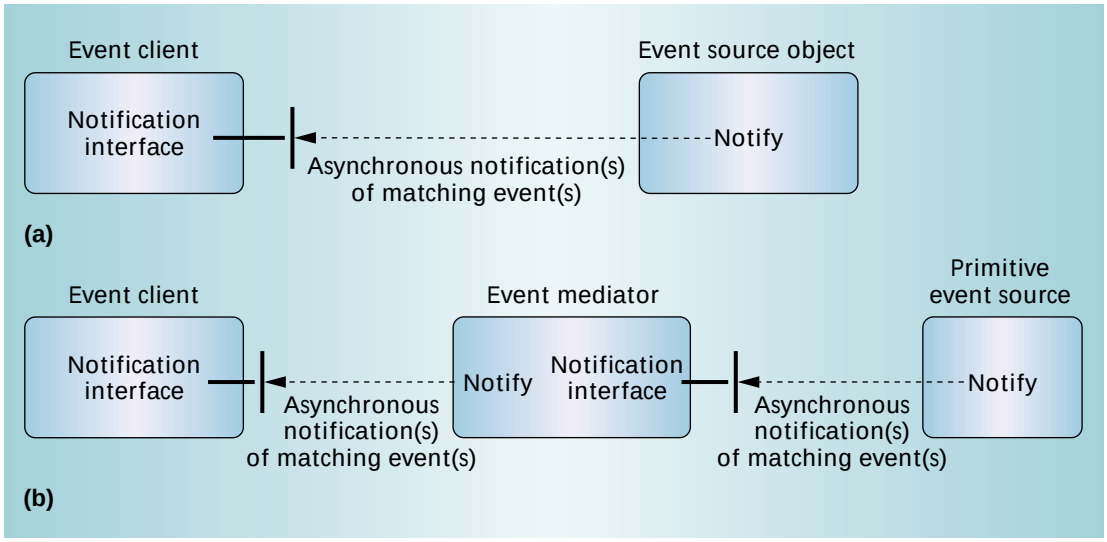

Figure 2. Event notification: (a) direct and (b) mediated. The event mediator can provide a higher level interface than the primitive event source(s), and it can register with any number of event sources on behalf of any number of event clients.

IDL the events it will notify to clients if asked, in addition to publishing its interface in the standard way.

Registration

In CEA, such an object has a *register* method in its interface, and interested parties can register interest in any event class, specifying a value or wild card for each parameter. CEA defines event occurrences as objects of a specified type. Therefore, CEA users can name and parameterize events at a fine granularity, using the full range of IDL types, with event notification at this granularity.

Access control comes into play at event registration. The service does not allow a client without appropriate authority to register, and events that the service will notify are subject to restriction by parameter value. When an event occurs, the service matches it against a stored template associated with each registration. Subject to access restrictions, each client whose template matches is notified of the event. Figure 1 shows the publish-register-notify scheme. (Chaoying Ma and Jean Bacon describe how this event extension was added to a CORBA implementation.[4]) CEA provides event functionality as part of any object's interface, whereas other event schemes are independent services.[5,6]

The publish-register-notify paradigm facilitates direct source-to-client event notification. Clients can request sources to filter events by supplying parameter values on registration. In addition, application developers can define intermediate services, sometimes called event mediators. Figure 2 shows the two forms of event notification. This flexible architecture allows developers to build any service structure.

Mediation

One use of a mediator is to remove the filtering function from a primitive event source by providing an indirection between the source and its potential clients. A source that cannot afford the overhead of template matching can notify such a mediator of all its detected events. An example is an active badge sensor, which detects badge identifiers and notifies all sightings to a higher level, the active badge service (in the mediator role). Instead of presenting clients an interface with parameters such as *badge-id* and *sensor-id*, the active badge service notifies events such as *seen*(*person*, *room*).

A mediator can also prevent a mobile user from missing events of interest while disconnected from the networked systems. The mediator registers interest with the required event sources on behalf of the mobile client and buffers the event notifications it receives from these sources. It also registers interest in the mobile client's location, and notification of an *attach* event (detecting the mobile user) triggers delivery of the accumulated events to the user at the new location. Another way to program this delivery is for the reconnected user to poll the mediator via its synchronous interface.

A mediator can provide functionality equivalent to the CORBA event service,[1] which registers interest in all notifiable events with event sources and provides its clients both a synchronous "pull" interface and an asynchronous "push" interface. OMG's recently specified notification service enhances the event service by defining structured events and providing filtering at the event service.[5] Implementing the notification service over CEA would be relatively easy, but making it efficient and scalable would require significant effort.

Our scheme scales well because it allows source-side filtering rather than requiring filtering at a centralized server or at the client. Other schemes multicast events and filter them at the client,[7] putting an unnecessary load on the communications infrastructure and clients. Consider the example of tracking users worldwide. It is unacceptable for all clients to receive notification of every event about every user, even from a single domain. With source-side filtering, only clients

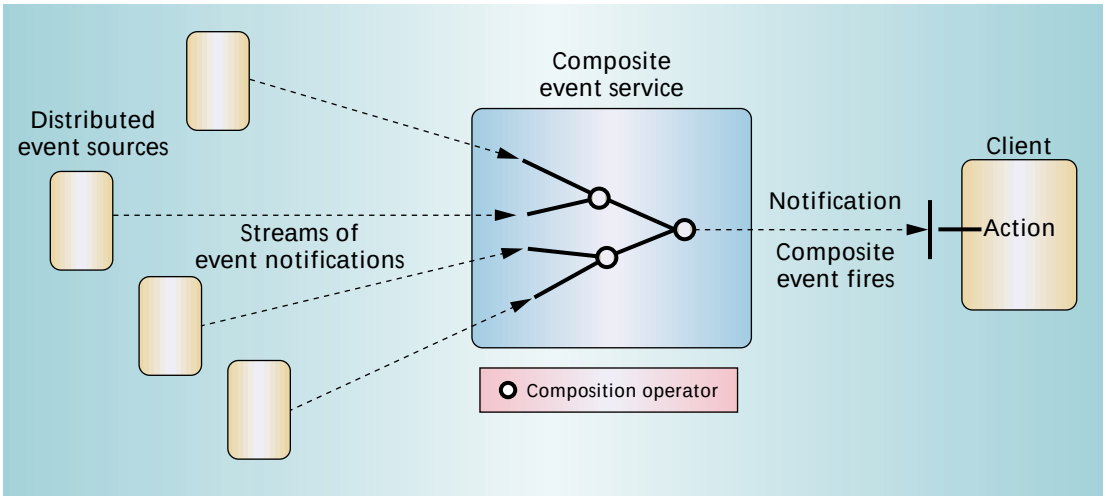

Figure 3. Composite-event detection: The client registers interest in a composite event with the composite event service (CES). The CES registers with the individual event sources, which then notify streams of events to the CES. The CES detects when the required event combinations occur and notifies the client of each composite event occurrence.

that have registered interest in a specific, parameterized event receive notification of that event. This fine-grained registration is the only way to keep network costs and client filtering at an acceptable level. The search complexity of template matching is limited by the number of parameters and therefore scales well with the number of subscribers, as well as minimizing the use of communications bandwidth.

Composite events

An extension of the simple, basic event paradigm combines primitive events into composite events. With this extension, developers can build composite-event services that will register their clients' interest with appropriate event sources and notify clients of composite events. A composite-event server is an event mediator that can perform filtering across events of different types from different sources.

There are many possible application scenarios for composite-event services. For example, a medical practitioner can ask to be notified if a patient's temperature rises by more than one degree within 24 hours of the administration of a new drug. Researchers can ask an active badge server to notify them when two or more members of their group are in any of the lab's meeting rooms on any Friday after 11 a.m. A composite-event service can detect faults across different component types in a communications network.

In CEA, we have defined composition operators and provided a language for specifying composite events. We use stream semantics to model event arrival from the various sources that make up distributed systems. Figure 3 illustrates the composite-event detection process.

OASIS

Oasis allows each service to define access rights for categories of users. A service can refer to both its own categories and those of other services. While using a service dynamically, a client holds a certificate that proves its right to do so, which the issuing service can revoke at any instant as a result of interservice notification.

Each Oasis service is responsible for classifying its clients into named roles:

- A login service defines the *logged-in-user* role with parameters such as *user-id* and *machine-name*.
- A patient-monitoring service defines roles such as *surgeon*, *doctor*, *nurse*, and *patient* with appropriate parameters.
- An examinations service defines roles such as *candidate*, *examiner*, and *chief-examiner*.
- A digital library service defines roles such as *reader*, *librarian*, and *administrator*.

Oasis provides a role definition language (RDL) in which services can specify precise conditions for clients to enter each role. RDL is a formal logic based on Horn clauses. A client gains authentication from a service by presenting it credentials that prove the client conforms to its policy for entry to a particular role. For example, to access an online examination in computing, you must certify securely that you are registered in the computing course, have paid your fees, and are logged in at a computer in the correct location at the correct time. Figure 4 shows the authentication process.

A service issues an authenticated client a role membership certificate (RMC), which the client presents with subsequent requests to use that service. An RMC, shown in Figure 5, is an encryption-protected capability that includes the role name, any parameters associated with that role, and a reference to the issuing service.

To become a candidate for the online computing exam, you must acquire an RMC for the named role *candidate*. The examinations service specifies in RDL that to achieve this you must present your *registered student* certificate issued by the registry, your *fees-paid* RMC issued by the accounts service, and your *logged-in-user* (*user-id, machine-name*) RMC issued by the

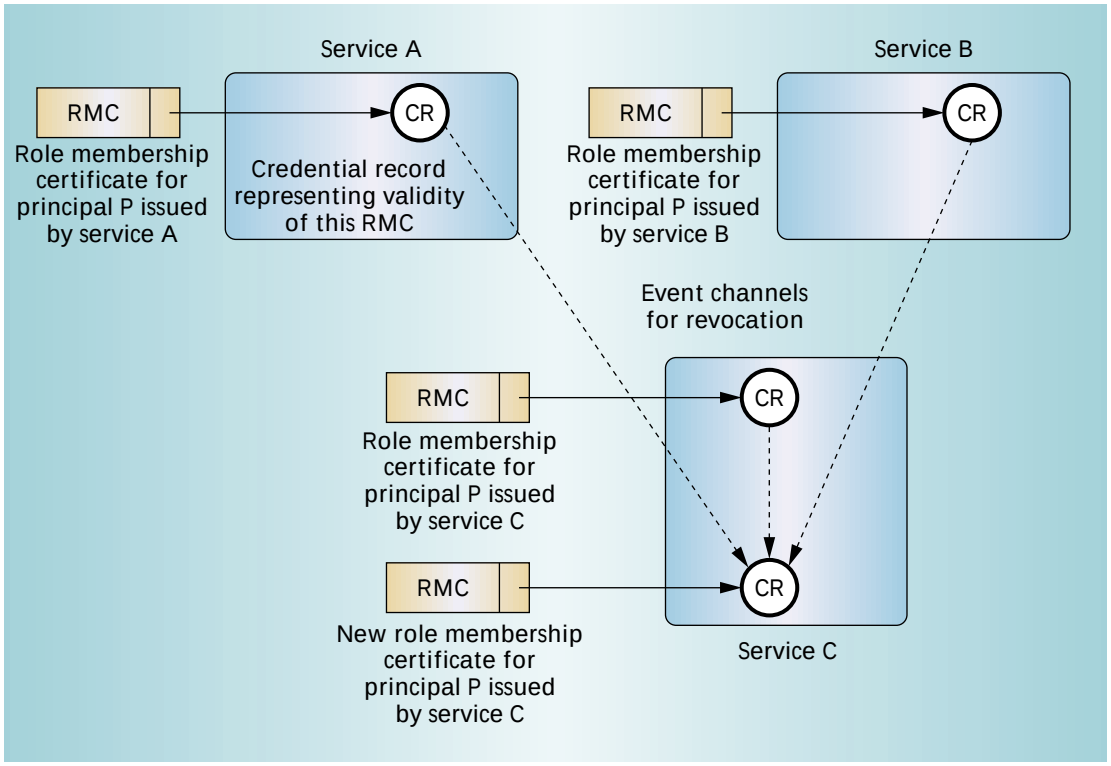

Figure 4. Entering a role. P presents credentials for entering a role of service C. These credentials are RMCs for specified roles of services A and B and an RMC for a different role of service C. C verifies the RMCs with services A and B and registers with them in order to be notified on revocation. C then issues P an RMC for the new role.

login service. The examinations service can check the local time. You are then ready to take the online exam. You present your *candidate* RMC with your request to see the exam questions.

An Oasis service maintains a credential record associated with each RMC it issues. The service uses the credential record reference (CRR) in an RMC (Figure 5) to locate the corresponding credential record. A proof rule of one service can refer to an authenticated user of another; that is, an RMC issued by one service may be required as a credential during authentication by another. The authentication process sets up a data structure embodying the proof. This data structure spans the services involved, linking the related credential records. The data structure is a dynamically maintained proof tree that exhibits, among other things, the trust relationships between the services in which the client has entered named roles. An earlier publication specifies RDL and describes the data structures that maintain the dynamic proof tree in detail.[8]

Comparison with other capability schemes

There are many encryption-protected capability schemes for use in distributed systems. The basic idea of these schemes is like the scheme we've described for Oasis—the issuing service computes and checks a signature to include as a field of a capability.

The Oasis scheme differs from traditional capability-based access control in several ways. RMCs are capabilities that implement role-based access control; the role is a protected field. Oasis is principal-specific; that is, the signature depends on the principal's identity as well as on the protected RMC fields. The RMC is therefore protected from theft by other principals—through network tapping, for example. Li Gong proposed that a persistent principal name, such as a user ID, should be a protected field of a capability.[9] Using a session-based principal ID, assigned at login, gives greater security than using a persistent principal name. Some applications require that the principal using a certificate be anonymous. Oasis achieves this simply by omitting the principal ID as an argument of the encryption function on certificate issuing and checking, since it is role membership that conveys access rights.

A more novel difference is that Oasis maintains RMCs dynamically. The RDL specification for acquir-

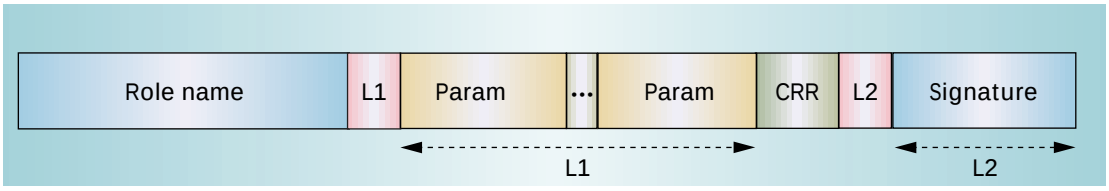

Figure 5. Role membership certificate (RMC). An Oasis service computes the signature field by applying an encryption function to the RMC fields that must be protected from tampering, the identity of the principal to which it was issued, and a secret random number held by the issuing service. The CRR is the credential record reference to the issuing service and the credential record within that service.

The format of RMCs is quite general, since they are intended for use by many interworking services that require different numbers and types of parameters.

ing an RMC also indicates which conditions must remain true for the RMC to remain valid. Should such a condition become false, this fact is signaled immediately to the RMC-issuing service, which instantly invalidates the RMC. Figure 4 shows the event channels set up to signal such violations. Thus, returning to our exam example, let's say you have been sent down in disgrace and are no longer a registered student. The registration service not only revokes your *registered-student* certificate, it also signals the examinations service, which immediately revokes your *candidate* RMC and your access to the exam questions.

Other capability schemes have not achieved this selective and immediate revocation at the granularity of a single capability. Revocation at the granularity of individual certificates makes sense primarily for principal-specific capabilities, and such schemes are uncommon. Immediate revocation requires a callback mechanism that includes failure detection, such as that provided by Oasis's heartbeat event management, which we describe later. Lightweight and selective revocation is possible only for appropriate architectures with system support in place.

Oasis-aware services

Some services (Oasis role-servers) define Oasis roles, and some do not but are Oasis-aware, or Oasis-conforming. A service of the latter type may wish to control which principals can use it in terms of RMCs issued by Oasis role-servers. This applies to both synchronous invocation and asynchronous registration and notification. Any service can specify that a principal present an RMC as a parameter of synchronous method invocation. It can also specify that a principal present an RMC as a parameter of the *register* invocation. For example, the active badge service can ensure that the vice-chancellor's whereabouts is revealed only to administration members who hold the role *secretary*.

A service can impose access control at both registration and notification. For example, if you register `seen(*, conference-room)`, which means "Tell me everyone who enters the conference room," the service might allow you notification of some matching instances but not others. A service can ask an RMC-issuing service for an RMC validity check at any time.

The format of RMCs (Figure 5) is quite general because they are intended for use by many interworking services that require different numbers and types of parameters. Each service is free to use a different mechanism for signing its certificates. This allows service designers to make appropriate trade-offs in signature length, computational cost, and security.

An Oasis-aware service wishing to verify an RMC must present it to the issuing service. To ensure that the certificate has not been tampered with, the issuing service checks the principal's identity, in a secure manner, and then validates the signature. Next, the issuing service follows the credential record reference, which identifies a record that indicates whether the certificate has been revoked. Finally the issuing service informs the Oasis-aware service of its decision.

The integrity and revocation checks use different techniques. An Oasis-aware service can cache the fact that the cryptographic check at the issuing service was successful. If the client presents the same certificate later, the service need only validate the credential record. Indeed, it can instead ask the issuing service to notify it of any state change, provided that the application runs on an asynchronous platform. This approach considerably reduces network traffic, as well as the access latency of authorization checks. The reduction is particularly significant when two Oasis servers communicate—for example, when certificates issued by one service depend on the validity of certificates issued by another.

Auxiliary credentials

Clients present RMCs on behalf of authenticated principals, typically processes associated with logged-in users. In many cases, the right to enter a role is long-lived; for example, a student might attend a university for several years. For this reason, Oasis also supports auxiliary credentials, which, like traditional capabilities, can persist across sessions. Oasis manages auxiliary credentials with two additional classes of certificate, the auxiliary credential certificate (ACC) and the revocation certificate. Role entry policy expressed in RDL can require the presentation of an ACC in addition to one or more RMCs. The ACC may give details of the additional RMCs required and the constraints on their parameters. An ACC can therefore extend the role entry policy stored at the Oasis issuing server.

A service can issue an ACC to a principal that presents appropriate credentials. For example, a student taking an online examination must present a *registered student* ACC. To obtain an ACC for the student, some principal must enter the role *registrar*. The principal can then apply for an ACC specific to the student's registration number. The service issues the ACC and creates a credential record. In addition to the ACC, the principal receives a revocation certificate, which contains a CRR for the *registered student* ACC and the role name *registrar* that created it. The principal now hands over the ACC to the student. Subsequently, any principal holding the role *registrar* can use this revocation certificate to invalidate the ACC that it references. In a recent paper, John H. Hine et al. describe the use of auxiliary credentials in more detail.[10]

DISTRIBUTED IMPLEMENTATION ISSUES

A fundamental characteristic of distributed systems is that nodes and links can fail at any time. Components of a distributed software system may fail or become unreachable but continue to work, and we cannot be certain which is the case.

Recall that Oasis servers cooperate dynamically to prove that authentication conditions remain true. The absence of a condition violation notification from one Oasis server to another means either that the condition is still true (all is well) or that a failure has occurred in the relevant service or its connection. In that case, the condition might have become false, but notification cannot be sent or received. For this reason, we use a heartbeat protocol between the Oasis servers involved. In the absence of an expected heartbeat, each server can specify its own policy. A server with high security requirements might suspend the affected RMCs until heartbeats resume.

Oasis uses service-to-service communication channels. For scalability, we can batch RMC validation requests at times of high client activity. This is a standard trade-off between system bandwidth demands and client-side latency. Another factor affecting the number of interdependent services that Oasis can support is the problem of managing application complexity. Interdependent services use one another's certificates, so their access control policies are interdependent and must be coordinated. The structure of certificates issued by service X does not concern service Y—only the role and possibly the parameter values. Because the dictates of simple design and security restrict the number of links for a given service, Oasis can realize the advantages of batching validations for large numbers of clients.

We determine the pulse rate of heartbeats on a service-to-service basis. We use a fast heartbeat when detecting failure quickly is important. Messages that assert continuing life are necessary only when there is no other communication between two services, and we can avoid the overhead of maintaining a heartbeat at times of high-bandwidth interaction.

Server failure can cause the loss of events waiting in main memory for transmission. Heartbeats alert the application to this possibility. If an application requires reliable transmission, the system must store events persistently so that it can recover them on restart. ODL-based automatic stub generation facilitates event storage in addition to transmission.

We continue our work on support for distributed applications in several areas: event storage and retrieval, composition, interoperability, and automation of access control policy specification.

Within CEA, it is easy to arrange that events be logged to an event store. The event store manager can register interest with event sources as described earlier and receive event notifications.[11] We have made transmitted and stored data types compatible by using CORBA IDL for transmitted data and ODL for stored data.

Clients can query the event store to retrieve events and event patterns. The problem of defining queries has much in common with that of defining composite events. Both involve applying a filter defined by a pattern to a collection of event occurrences. For example, a client wishing to replay aspects of a specified meeting asks, "Show me the whiteboard during the time when we paused the (name) video." We are also working on ways to add visualization to event replay. For example, in our laboratory, we have replayed active badge events by showing avatars moving through a Virtual Reality Modeling Language representation. We plan to use OQL for querying our ODMG-based event store and a commercial, ODMG-compliant database management system.

Current standards adequately support synchronous invocation, but some applications will also require asynchronous notification.

The practice of composing applications from reusable components is increasing. Developers need a mechanism that allows independently developed components to interoperate. Components must make their interface specifications available to clients, either by publishing them in a name server or by allowing clients to interrogate them. The ODMG object metadata interface allows a component to provide such a service. Current standards adequately support synchronous invocation, but applications such as the active home, the active office, and virtual and augmented reality will also require asynchronous notification. Event classification, publication, and notification provide a plug-and-play mechanism for the construction of distributed applications. John Bates et al. describe initial work on this style of active programming.[12]

Applications that function over a wide area will require systems to interoperate. Researchers are developing special-purpose protocols to achieve cross-domain and cross-platform interoperability. We see object and event class hierarchies as central to achieving interoperability. We are using ODL to specify event classes and generate the ODMG standard metaobject representation, including class specialization. For example, suppose we wish to track the movements of someone worldwide. All domains must agree that there is a class called *locate*. In some domains, this class specializes as *login* detection only; in others, it specializes as *active-badge-locate*; in still another, it specializes as iris recognition technology. We expect that registering interest at a high level will be sufficient to register interest at the specialized levels. Such a scheme would allow dynamic integration of new technology without wholesale recompilation of existing systems.

Nonexpert users of computer systems specify access control policy in natural language, whereas Oasis uses logical notation. It is vital that policy can evolve over time in a controlled way. We are investigating ways to express policy precisely and to check policies expressed at different services for consistency. We are evaluating Oasis through a case study involving networked electronic health records. ✷

Acknowledgments

We acknowledge the support of the UK Engineering and Physical Science Research Council (EPSRC) through grants GR/J42007 (interactive multimedia presentation support) and GR/K77068 (active systems), the latter in collaboration with Nortel Technology. Thanks also to ICL for supporting our research. Thanks to past and present members of the University of Cambridge Computer Laboratory Opera Research Group and to John Hine for their contributions. We are in debt to the anonymous reviewers who encouraged us to sharpen the discussion of several systems issues.

References

1. *The Common Object Request Broker: Architecture and Specification, Revision 2.0*, Object Management Group, http://www.omg.com, July 1995.
2. R.G.G. Cattell, *The Object Data Standard ODMG 3.0*, Morgan Kaufmann, San Mateo, Calif., 2000.
3. D. Barry and T. Stanienda, "Solving the Java Object Storage Problem," *Computer*, Nov. 1998, pp. 33-40.
4. C. Ma and J. Bacon, "COBEA: A CORBA-Based Event Architecture," *Proc. 4th Usenix Conf. Object-Oriented Technologies and Systems (COOTS 98)*, Usenix, Berkeley, Calif., June 1998, pp. 117-131.
5. *Notification Service*, OMG TC Document Telecom/98-06-15, Object Management Group, Framingham, Mass., June 1998.
6. R.E. Gruber, B. Krishnamurthy, and E. Panagos, *READY: A Notification Service for Atlas*, tech. report, AT&T Labs-Research, Florham Park, N.J., 1997.
7. K. O'Connell and V. Cahill, "System Support for Scalable Distributed Virtual Worlds," *Proc. ACM Symp. Virtual Reality Software and Technology*, ACM Press, New York, 1996, pp. 141-142.
8. R. Hayton, J. Bacon, and K. Moody, "OASIS: Access Control in an Open, Distributed Environment," *Proc. IEEE Symp. Security and Privacy*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 3-14.
9. L. Gong, "A Secure, Identity-Based Capability System," *Proc. IEEE Symp. Security and Privacy*, IEEE CS Press, Los Alamitos, Calif., 1989, pp. 56-63.
10. J.H. Hine et al., "An Architecture for Distributed OASIS Services," *Proc. Middleware 2000, Lecture Notes in Computer Science*, Vol. 1795, Springer-Verlag, Heidelberg and New York, 2000, pp. 107-123.
11. M. Spiteri and J. Bates, "An Architecture to Support Storage and Retrieval of Events," *Proc. Middleware 1998, IFIP Int'l Conf. Distributed Systems Platforms and Open Distributed Processing*, Int'l Federation for Information Processing, Geneva, 1998, pp. 443-458.
12. J. Bates et al., "Integrating Real-World and Computer-Supported Collaboration in the Presence of Mobility," *Proc. IEEE Workshops on Emerging Technologies in Collaborative Environments 1998, Workshop on Collaboration in the Presence of Mobility*, IEEE Press, Piscataway, N.J., 1998, pp. 256-261.

Jean Bacon is a reader in distributed systems at the University of Cambridge Computer Laboratory. She also leads the Opera Research Group with colleague Ken Moody. Contact her at jmb@cl.cam.ac.uk. For information about the Opera Group, see http://www.cl.cam.ac.uk/Research/SRG/opera/.

Ken Moody is a lecturer in computer science at Cambridge University and co-leader of Opera. Contact him at km@cl.cam.ac.uk.

John Bates is a lecturer in the Laboratory for Communications Engineering, University of Cambridge, Department of Engineering. During the work described in this article, he was a member of Opera. Contact him at jb141@eng.cam.ac.uk.

Richard Hayton now works at Citrix Systems, Cambridge. He was previously a member of Opera at the University of Cambridge. Contact him at richard.hayton@eu.citrix.com.

Chaoying Ma is a research associate in the Opera Group at the University of Cambridge. Contact her at cm@cl.cam.ac.uk.

Andrew McNeil is a PhD candidate in the University of Cambridge Computer Laboratory and contributed to the Opera work described here. Contact him at aam1005@cl.cam.ac.uk.

Oliver Seidel is a PhD candidate in the University of Cambridge Computer Laboratory and participated in the Opera project described here. Contact him at os10000@cl.cam.ac.uk.

Mark Spiteri now works in the Laboratory for Communications Engineering, University of Cambridge, Department of Engineering. He was formerly in the Opera Group. Contact him at mds24@cam.ac.uk.