Proceedings of the Eighth IASTED International Conference
SOFTWARE ENGINEERING AND APPLICATIONS
November 9-11, 2004, Cambridge, MA, USA

UTILIZING COMMERCIAL OBJECT LIBRARIES WITHIN LOOSELY-COUPLED, EVENT-BASED SYSTEMS

Jie Ren, Richard N. Taylor
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
+1 949 824 2776
{jie, taylor}@ics.uci.edu

ABSTRACT

A class of loosely-coupled, event-based systems is emerging as an effective candidate solution for constructing large, heterogeneous, and dynamic software architectures. Given the dominancy of classic object-based remote procedure call-based applications, it is important to bridge the gap between the two paradigms so the past investment can be preserved and the transition to the new model can be eased. This paper addresses the issues encountered in utilizing a commercial object RPC library, namely Microsoft Component Object Model, within event-based systems. A framework is proposed to integrate legacy applications into event-based systems. Its effectiveness is demonstrated through an experiment to integrate a commercial COM product with an event-based development environment. Some problems worthy of further research are also identified.

Keywords

Software Architecture, Event-based System, Component Object Model, Integration,

1. INTRODUCTION

Software architecture has been proposed as an effective solution for producing bigger, better and cheaper software [1]. The research community has not reached a universally accepted agreement about the concepts and terminologies used in software architecture yet. However, some more commonly proposed definitions are: a software system is composed of components and connectors, components are loci of computation and connectors are loci of communication, and a specific set of components and connectors form the configuration of the software [2].

The key of composition and integration in architectural-driven component-based development lies in connectors. Many technologies have been used in connectors, such as pipe-and-filter, remote procedure call, and object request broker. The variants have different capabilities and limitations [3]. Among them, the loosely coupled, event-based connection looks promising in integrating concurrent, heterogeneous components in dynamic environments [4]. In this paradigm, components communicate with each other by sending events (or messages. These two terms are used interchangeably in this paper), while connectors provide the infrastructure for messaging, including event registration, routing, and monitoring. The components can be written in different languages, reside on different processes, and run on different machines. They don't need to maintain specific pointers about the components that they are communicating with, and they can be easily added or removed from the system without adversely affecting other members.

The majority of current software systems are designed without much formal treatment of their architectural aspects. They are constructed using different flavors of procedural-calls. The procedure call can be local or remote, traditional or object-oriented. Given the vast amount of the legacy software, it is important to provide a bridge between these systems and the newly emerging event-based systems. In addition to the direct advantage of reusing the legacy, a bridge can facilitate evolution of these legacy systems so they can be refactored to benefit from research results about software architecture and event-based systems. Since the area of software architecture and event-based systems is still in active research, the bridge will also provide insights on finding effective solutions to constructing large and complex software systems.

This paper investigates issues and possible solutions in utilizing commercial reuse frameworks within loosely-coupled, event-based systems, with a motivating problem of integrating a commercial object library into an event-based software architecture development environment. Section 2 gives a general introduction to event-based systems. Section 3 surveys the commercial library, Microsoft COM. Section 4 discusses the key issues in integration. Section 5 outlines one possible solution addressing these issues and demonstrates its application. Section 6 concludes the paper.

2. EVENT-BASED SYSTEMS

There have been many research and commercial event-based systems. An early research work is Field [5]. OMG and Sun has proposed standards on event services for CORBA and Java, respectively. A framework capturing many relevant design dimensions for event

observation and notification systems was proposed by Rosenblum and Wolf [6]. The framework comprises seven models. An *object model* characterizes events generating components and notifications receiving components. An *event model* precisely identifies the phenomenon of an event. A *naming model* defines how a component refers to other components and events. An *observation model* defines mechanisms for observing and relating event occurrences. A *time model* describes the temporal and causal relationships between events and notifications. A *notification model* defines mechanisms used by components to subscribe and receive notifications. Finally, a *resource model* defines where the observation and notification computations are located in the Internet and how resources for the computations are allocated and accounted.

SIENA [7] is an Internet scalable event notification service. As an infrastructure service, it provides two primary services to components: notification selection (i.e., determining which notifications match which subscriptions) and notification delivery (i.e., routing matching notifications from publishers to subscribers). SIENA tries to achieve the expressiveness of the event selection mechanism without sacrificing the scalability in the delivery mechanism. Its notification selection service provides several interfaces to enable richer expressiveness: *unsubscribe*, *advertise*, and *unadvertised*, in addition to standard interfaces such as *publish* and *subscribe*. SIENA provides a language to express the filters and patterns that are used in event selection and delivery. Its notification delivery service supports hierarchical architecture, acyclic peer-to-peer architecture, general peer-to-peer architecture, and hybrid of the above architectures. The routing strategy of SIENA is similar to that of IP multicast, sending a notification only toward event servers containing clients that are interested in that notification. The strategy applies the same rule to patterns of notifications.

KnowNow [8] is an Internet-scale event service. It provides the following key functionalities: asynchronous event-driven operations; dynamic routing, filtering, and transformation rules; reliability and security across applications and services; and multi-protocol compatibility. Its architecture contains four components connected by HTTP: event routers, router modules, microservers, and gateways. The event routers manage topics, events, and subscribers, forming a network of networks to accomplish Internet scalability. The router modules perform functions such as data transformation, content filtering, and authentication/authorization. The microservers are optional client-side components that manage persistent, bi-directional connections between an application and an event router. The gateways connect KnowNow with other infrastructure technologies, such as message queuing, email and database. KnowNow provides reliable communication through store-and-forward event routers. It uses URLs for topic names and adopts HTTP as the communications protocol, allowing HTTP-aware applications to publish events without modification. KnowNow's topic routing and flexible subscriptions enables dynamic data aggregation and syndication. Its content-based routing and transformation allows flexible and powerful content distribution across organizational boundaries.

C2 is an event-based architecture style [4]. Its basic tenets include: 1) Components communicate with each other only by sending events; 2) Connectors route events; 3) Components and connectors both have one top interface and one bottom interface. (Hereon components and connectors are collectively referred as bricks.); 4) Components and connectors are connected to form a layered topology; 5) Components can be connected to at most one connector at any of its interfaces, while connectors can connect to any number of components and connectors at any of its interfaces; 6) Bricks send *request* events to upper bricks for service, and the upper bricks reply by sending *notification* events downwards. From the viewpoint of publish/subscribe, bricks publish requests to the upper ones and notifications to the lower ones, meanwhile they subscribe to requests from downward bricks and notifications from upward bricks.

These systems demonstrate the power provided by the current generation of event delivery services and the emergence of a new approach of constructing large complex software. However, few of the services address the problem of bridging between the event-based systems and the legacy world.

3. OBJECT-BASED REMOTE PROCEDURE CALLS

During the past decade, object-based remote procedure call has been getting wide acceptance in developing distributed software systems [9]. One prominent candidate for the core middleware, Object Request Broker, is the Component Object Model (COM) technology from Microsoft. The technology evolved from a desktop integration solution into a set of powerful services for the Windows operating system.

The core concepts of classic COM are as follows [10]: *interface, class, object, and apartment*. An *interface* is a set of functions. Each interface is designated by a Global Unique Identifier (GUID). An interface is immutable after its publication.

A *class* implements a set of interfaces. It is also designated by a Global Unique Identifier. An *object* is an instance of a class. When a client needs service from an object, it can get a reference to the object through either instantiating a new instance or getting an existing reference using a registry. The client can inquire an object whether it supports a specific interface.

The *apartment* is COM's mechanism for threading and marshaling management. A single thread apartment can host only one thread and the objects created by that thread, while a multiple thread apartment hosts many threads and the objects created by those threads. Whenever an object in one apartment calls an object in

another apartment, special marshaling is performed to achieve the location transparency.

COM also provides an *automation* facility to declare and determine type information, enabling clients to discover and invoke objects during run-time.

COM uses a *source interface* as a reverse communication channel from objects to clients. A source interface is declared in the object, but is implemented by the client. When an object processes a call from a client, it can invoke functions in the source interface and call back to the client.

New value added services, such as transaction, activation, and security, adds a *context* to the core concepts of the COM technology. A *context* provides a controlled environment under which the regular procedure calls are monitored and augmented to provide the required semantics. It needs just declarations of the desired properties from object developers to provide the service.

4. ACCIDENTAL AND ESSENTIAL ISSUES IN INTEGRATION

In his classic paper [11], Brooks divided the difficulties of software technology into *essence*, the difficulties inherent in the nature of software, and *accidents*, those difficulties that attend software production at the time being but are not intrinsic. Following him, we divide the difference between software systems that are based on traditional commercial reuse frameworks and systems that are based on events into essence and accidents. We argue the platform and language issues are accidental, and the essential issues lie in how the system is constructed.

4.1 Accidents

We believe that the essential difference between traditional commercial reuse frameworks and event-based systems does not lie in the platform issues. The capability of crossing the boundaries between processes, machines, operating systems, and network protocols is not unique to event-based systems.

COM is a good example to illustrate this. COM debuted as a desktop integration technology to connect different processes running on the same computer, but it is designed from the start to be able to run across different computers, and its wide deployment has confirmed this. While some people argue COM is a Windows-only technology, Microsoft tried to provide other operating systems with ported COM implementations. They even established a standard group to oversee the port. The failure of this maneuver is not because of technical reasons. COM's application level protocol, Object RPC, can run on top of a set of transport protocols, including HTTP.

We believe programming language issues are not essential to the difference between traditional commercial reuse frameworks and event-based systems, either. COM can be programmed using a set of languages, such as C++ and Basic. Its binary layout definition and the aforementioned automation support enable both early-binding programming languages and late-binding scripting languages to access functionalities of COM objects. Microsoft even provided a run time environment to support Java. Another object middleware, CORBA, has language mapping definitions allowing many choices among languages.

Jintegra [12] is a good example of the accidental nature of platforms and programming languages. It is a pure Java implementation of COM, enabling a COM object to be accessed as a Java object and a Java object to be accessed as a COM object. It can run on any Java-enabled platforms, including UNIX systems. This proves the issues about platforms and programming languages are not the differentiator between systems based-on traditional commercial reuse frameworks and system based on events.

4.2 Essence

We believe the logical architecture of a system is the essential difference between systems based-on traditional commercial reuse frameworks and system based on events. We argue the following two properties of event-based systems are the decisive differences: lack of explicit reference and asynchrony.

An explicit reference tightly couples a client and a server in traditional reuse frameworks. The client has to get a reference to the server first before it can send requests for services. Usually there will be a registry to store references for the available servers. Sometimes there are multiple levels of mapping so servers can be easily named. For example, an interface pointer is COM's core reference. It is usually obtained through instantiation using information stored in the Windows registry. Meanwhile, COM can create new objects using easy readable program identifiers and monikers, which are eventually mapped into class identifiers and interface identifiers. In traditional reuse frameworks, the binding of the reference can be either early binding, using information known before run time, or late binding, using information discovered during the execution.

Event-based systems, on the other hand, are loosely coupled. An event publisher generally does not know to whom the events will be delivered. The subscriber can subscribe to many types of events, regardless of the origins of the messages. The coupling between components is greatly reduced compared to traditional systems.

Traditional commercial reuse frameworks are also mostly synchronous. The client thread executes in a blocking manner. After sending a request, it waits for the reply before moving to the next step. This procedural paradigm lies deeply in the architecture of existing systems.

Event-based systems are asynchronous. After a publisher publishes an event, it moves to the publication of the next event. A subscriber handles the arrival of its subscriptions in a non-blocking manner, too.

We believe the above two issues in the architectural construction are essential to the difference between the systems that are based on commercial reuse frameworks and event-based systems. They are critical in bridging the two types of systems.

5. THE EXPERIMENT

During our research of utilizing the commercial object library in event-based systems, we first explored the built-in support of the commercial library and found it is unsatisfactory. We then designed a new framework for bridging the two paradigms. We validated the new framework by integrating a commercial COM product and a research event-based architecture development environment.

5.1 Limitation of the built-in support

The COM technology has evolved into a system supporting more than just pure object-based remote procedure calls. It can now automatically generate stubs for asynchronous calls from a single IDL definition. It introduces an event service, and provides a Message Queue system allowing a certain degree of message delivery.

The problems with these built-in features are they are rather rudimentary, providing limited functionality compared to a full-scale event-based system. The first problem is that the event in COM event service is just an interface definition. It has to be defined before the running of the application. Currently there is no mechanism to define a new type of event during run-time and allowing the application to discover its properties. The second problem is that the event service provides no routing. An event is always delivered from a publisher to a subscriber subject to filtering specified by the publisher and the subscriber. The two problems result in an architecture unsuitable for a large, complex system.

5.2 The bridging framework

Because of the limitations of the built-in event support in the COM technology, we decide to design our own solution. To bridge the COM library and event-based systems, we design a set of interfaces and provide several classes that implement those interfaces. These interfaces expose the conceptual model of event based-systems, namely components, connectors, events, and event handlers. They are COM-compatible, and can be used by any COM developers to bridge their applications with event systems. They also try to address the essential issues in integration, namely the coupling of object references and the synchronous nature of interaction.

We define a COM interface IEvent. It encapsulates the concept of an event. It provides a set of methods to manipulate and inspect properties of an event, such as the sender, the receiver, the type, the identifier, the time-stamp, and the content.

We define another COM interface IComponent. This interface represents a component in an event-based system. The main function of the interface is HandleEvent. A COM component can implement this interface to handle incoming events, mostly by computing and emitting outgoing events through connectors connected to it.

We also define a third COM interface IConnector. One main function of this interface is HandleEvent. The namesake function in the IComponent interface handles events in a computation-centered manner. However, the IConnector interface provides the communication capability of event-based systems. It provides event routing and filtering. It either delivers the event to the attached IComponent if the event matches the subscription of that IComponent, or it routes the event to the next available IConnector for further transportation. The IConnector interface also provides functions to attach an IComponent or another IConnector to the connector, and to detach an IComponent or another IConnector from the connector. These functions can be used to configure a system by connecting components and connectors. A third category of functions of the IConnector interface is used by an IComponent to subscribe events and publish events to the IConnector.

The interface of IConnector is summarized as below, using COM's Interface Definition Language:

```
Interface IConnector {
  HRESULT HandleEvent(IEvent *);
  HRESULT Attach(IComponnet *);
  HRESULT Detach(IComponent *);
  HRESULT Attach(IConnector *);
  HRESUTL Detach(IConnector *);
  HRESULT Publish(IEvent *evt, IComponent *pub);
  HRESULT Subscribe(IEvent *pt, IComponent
*sub);
}
```

The framework provides several classes that supply basic implementations of these interfaces. The CEvent class, the CComponent class, and the CConnector class implement the IEvent interface, the IComponent interface, and the IConnector interface, respectively. The CEvent provides enough functionality of the IEvent interface. The CComponent class is a dummy class. Developers need to implement the IComponent interface in their own components. The CConnector class provides basic event delivery using standard sockets. All the concepts of event-based systems are expressed using standard COM interfaces, functions, and classes. These COM compliant constructs can be accessed by any COM development environment.

In this framework, there is no explicit reference between two components. A component only knows its neighboring connectors. It attaches itself to the related connectors, and publishes interesting events to the connector. Connectors decide what the architecture topology of a system is and where the events are routed. When a COM application uses the framework to send an event, a publisher COM component implementing the IComponent interface makes a regular COM call to a COM connector implementing the IConnector, passing the event implementing the IEvent interface as the parameter. The connectors use any suitable transport technology to eventually route the events to the connector

that is connected to the subscriber component. From there, the connector makes the final call on the subscriber and delivers the event for handling. Viewed at a micro level, the process is completely COM compatible and all interactions are COM method invocations. Viewed at a macro level, the process conforms to a loosely coupled event style.

The basic functions in the interfaces and classes permit asynchronous message sending and handling. A component sends outgoing messages and handles incoming messages, without any blocking on the execution thread. However, the majority of COM applications are still synchronous. A typical COM application waits for the response of a request before it can move to the execution next. To easily integrating these synchronous applications, an additional `SendAndWait` function is provided in the IConnector interface. When a request is published through this function, the executing thread is blocked instead of returning immediately. A separate thread monitoring the incoming events will search for the response to the request just sent. Once the response arrives, the monitoring thread notifies the blocked thread, and the blocked thread resumes and returns with the response to the previously sent request. Thus, a blocking procedure call that returns results is achieved on top of the non-blocking event delivery mechanism.

5.3 The validation

To validate our research approach, we integrate a COM based commercial product into our event-based architecture development environment.

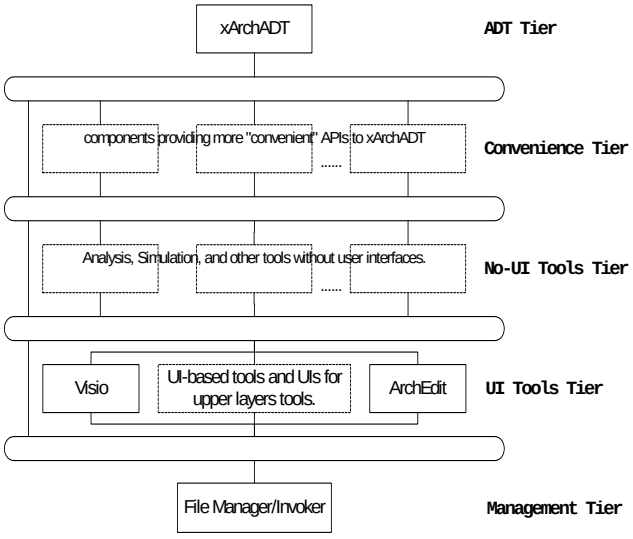

Figure 1, Architecture of ArchStudio

We choose C2 as a representative of the event-based architecture styles. We develop a software development environment, ArchStudio, to support the development of software in this style [13]. ArchStudio itself is in C2-style. Its architecture is depicted in Figure 1.

In Figure 1, rectangles designate components. Those in solid line rectangles are complete components, while those in dashed line rectangles are components to be added into the environment. Round angle boxes designate connectors. The core of ArchStudio is a component named xArchADT. It stores architectural information expressed in xADL 2.0, an extensible, XML-based Architecture Description Language [14]. A convenience tier provides easy access to this data repository. Some non-UI tools provide analysis, simulation, and monitoring capability. A set of UI tools is used to manipulate the architectural information graphically. An invoker provides a portal for accessing all integrated tools.

Most of the components and connectors in the environment are written using a Java-based framework that is to ease the construction of event-based, loosely coupled software. The framework supports components, connectors, interfaces, and messages.

The COM product integrated into the event-based ArchStudio is Visio, the diagramming solutions of Microsoft Office suite. Like other Microsoft products, Visio supplies a set of COM interfaces to allow programmatic access of its capability.

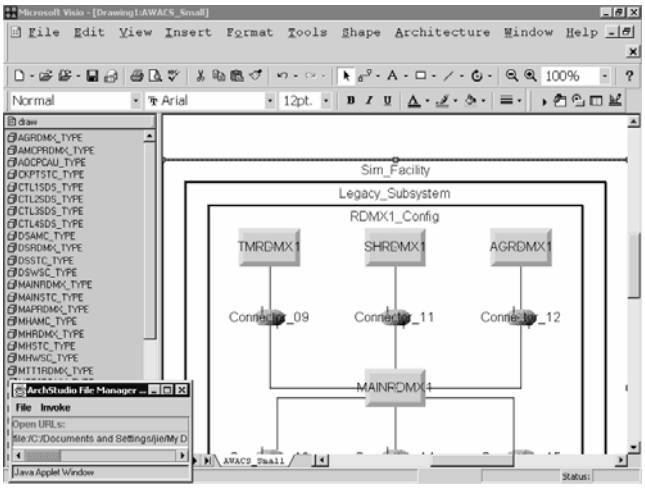

Figure 2, ArchStudio with Visio front end

Figure 2 shows the resulting environment that integrates Visio with other ArchStudio components [15]. The ArchStudio File Manager is the central portal to the various tools of ArchStudio environment. The main Visio window shows part of the architecture for AWACS (Airborne Warning and Control System). The architecture is described using xADL 2.0. The graphical layout is generated from this description with the help of AT&T Research's open-source Graphviz Dot tool [16].

In the integrated environment, Visio publishes requests to xArchADT for insertion or removal of xADL elements corresponding to components and connectors created or deleted by the architect. For example, when the architect creates a new component, Visio sends an insertion request to xArchADT, specifying the type and identifier of the new component. Visio also uses the `SendAndWait` API to retrieve information from xArchADT synchronously, such as the currently available component types and connector types.

In the meantime, Visio also subscribes to notifications when other components of ArchStudio publish. For example, when ArchEdit, a generic syntax-directed editor, requests deleting a connector, xArchADT fulfills the request and publishes a message about the removal. Visio gets the notification and removes the connector from its graphical display. ArchEdit will not delete any connected links of the deleted connector because of its simplicity and generality, but Visio enforces the correct semantics and deletes those links, publishing further requests to xArchADT to remove the elements corresponding to the newly removed links.

6. CONCLUSION

The loosely coupled, event-based system is promising in integrating concurrent, heterogeneous components in dynamic environment. It is important to utilize the current generation commercial reuse libraries within new systems, so the legacy software can be reused and evolved and the new style can be better received.

Our research is an initial effort in this direction. It provides a basis to allow legacy objects communicating with event-based systems and facilitates event-based systems communicating with those legacy objects. Our research tries to bridge the essential gap between the two paradigms, mitigating their differences in explicit references and synchronous operations.

The connector is in the center for architecture configuration and events routing. Our current framework only supports manually configuring connectors. Our view in architecture-based evolution envisions a model centered an evolution manager [17]. The manager maintains both a design time model and a run time model, accommodating system operation and evolution based on the explicit models. Currently the manager operates in a Java-based event system. We plan to extend the capability to the COM interoperation framework and bridge the two settings into a complete solution.

Our framework currently enables only basic communication between the legacy components and the new systems. How to integrate advanced functionalities into event-based systems is essential in achieving the full potential of the new paradigm. Among these functionalities, security proposes challenging research questions. Currently enforcing security depends on a tightly controlled environment. How to provide these functionalities in a loosely coupled, event-based system is worth further exploration [18].

7. REFERENCES

[1] D.E.Perry, A.L.Wolf, Foundations for the study of software architecture**,** *SIGSOFT Software Engineering Notes, 17*(4), 1992, 40-52.

[2] N.Medvidovic, R.N.Taylor, A classification and comparison framework for software architecture description languages, *IEEE Transactions on Software Engineering, 26*(1), 2000, 70-93.

[3] M.Shaw, D.Garlan, Software architecture: perspectives on an emerging discipline(Prentice Hall, 1996).

[4] R.N.Taylor, N.Medvidovic, K.M.Anderson, E.J.Whitehead, Jr., J.E.Robbins, K.A.Nies, P.Oreizy, D.L.Dubrow, A component- and message-based architectural style for GUI software, *IEEE Transactions on Software Engineering, 22*(6), 1996, 390-406.

[5] S.P.Reiss, Connecting tools using message passing in the Field environment, *IEEE Software, 7*(4), 1990, 57-66.

[6] D.S.Rosenblum, A.L.Wolf, A design framework for Internet-scale event observation and notification, *Proc. 6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on Foundations of Software Engineering*, September 1997, 344-360.

[7] A.Carzaniga, D.S.Rosenblum, A.L.Wolf, Design and evaluation of a wide-area event notification service, *ACM Transactions on Computer Systems, 19*(3), 2001, 332-383.

[8] KnowNow Architecture Overview, http://www.knownow.com/products/whitepapers/KnowNow_Architecture_Overview.pdf

[9] R.Orfali, D.Harkey, J.Edwards, *Client/server survival guide, 3rd edition* (John Wiley & Sons, Inc., 1999).

[10] G.Eddon, H.Eddon, *Inside COM+ base services* (Microsoft Press, 1999).

[11] F.P.Brooks, No silver bullet: essence and accidents of software engineering, *Computer, 20*(4), 1987, 10-19.

[12] http://www.linar.com/

[13] R.Khare, M.Guntersdorfer, P.Oreizy, N.Medvidovic, R.N.Taylor, xADL: enabling architecture-centric tool integration with XML, *Proc. 34th Annual Hawaii International Conference on System Sciences*, 2001, 9-17.

[14] E.M.Dashofy, A.van der Hoek, R.N.Taylor, A highly-extensible, XML-based architecture description language, *Proc. 2nd Working IEEE/IFIP Conference on Software Architecture*, 2001, 103-112.

[15] J.Ren, R.N.Taylor, Visualizing Software Architecture with Off-The-Shelf Components, *Proc. 15th International Conference on Software Engineering & Knowledge Engineering*, 2003, 132-141.

[16] http://www.research.att.com/sw/tools/graphviz

[17] R.Fielding, E.J.Whitehead Jr., K.Anderson, P.Oreizy, G.Bolcer, R.N.Taylor, Web-based development of complex information products, *Communications of the ACM, 41*(8), 1998, 84-92.

[18] C.Wang, A.Carzaniga, D.Evans, A.L.Wolf, Security Issues and Requirements for Internet-scale Publish-Subscribe Systems, *Proc. 35th Annual Hawaii International Conference on System Sciences*, 2002.