Flexible Distributed Event-Driven Programming Framework for Networked Appliances and Sensors

Masayuki Iwai¹, Jin Nakazawa¹, and Hideyuki Tokuda² ¹
¹Graduate School of Media and Governance, ²Faculty of Environmental Information, Keio University
5322 Endo, Fujisawa, Kanagawa 252-8520, Japan

Abstract

This paper presents a new framework for constructing flexible event-driven systems for distributed networked devices and appliances, referred to as Dragon. Dragon enables constructing and modifying an event-driven system easily by visualizing distributed services which are handled in an integrated manner. The update of the system is conducted immediately following a modification by Dragon. Dragon is built on Jini/Java and can be executed on any Jini/Java-compliant machines. In addition, Dragon incorporates soft real-time event delivery to enhance the predictability of behavior of the constructed system and composition of distributed services for event management. In this paper, we describe the details of design and implementation of Dragon.

1. Introduction

Over the past few years, we have seen an evolution of techniques for networked and Java-based appliances that are aimed for use in homes and offices. The age in which PCs dominate all other devices and networked appliances will be finished in a few years. As one aspect, network-ready hardware such as TINI[4] has accelerated this trend. Software for embedded devices such as light weight Java[16] and real-time Java[1, 11] facilitate combinations of the software that is distributed over a network. Furthermore, middleware such as HAVi [15], UPnP[10], VNA[12], and Jini[17] provide plug-and-play facilities. Although these hardware and software components have become available, appliance users are not sufficiently relieved from complications in combining the components.

We consider several issues towards binding appliances and devices more flexibly and easily. First, services, components software that constitutes appliances and sensors become unavailable due to failures in networks or replaced with better alternative components. These changes should be notified to a manager of appliances. In one case, a user desires to display a message that indicates that a washing machine has completed its job on a networked TV, while in another case, the user may want a nearby speaker to make sound when such an event has happened. A designing system needs to cope with this versatility. Second, these components are distributed over a home network or the public Internet. Components on the Internet are less reliable than those on the home network with respect to connectivity. In addition, a suitable convention of name resolution and service discovery is required. Finally, flexible delivery of events may be required for some systems. For example, a networked infrared detector of thieves must deliver an emergency event to an electric door to lock it immediately. Such systems need real-time notification, no matter how the system is overloaded. To consider use in home or office networks, too complicated system prevents users from having the benefit of networked appliances and sensors. Without a good designing system, a system programmer would have to pay attention to these complicated issues.

To overcome the above issues, we propose a programming framework which is called "Dynamic reconfigurable event delivering framework with smart generation (Dragon)." Dragon is a system created on top of Jini that can easily, improvisatory, and immediately configure relationships among events generated at distributed services.

Each element in Dragon does not have much powerful functions. However we aim to create ad-hoc in-
telligent system with all elements together. Dragon reduces both the programmer’s development cost and user’s combinational configuration cost by supporting a graphical editor for combining event paths between services. Dragon aims at being used not only by programmers but also by end-users at home with a graphical editor. Although the editor of Dragon is similar to JavaBeans Editor[18], it differs in that the actual constructed system is reconfigured immediately after it is edited on the Dragon editor and in that the Dragon aims at distributed services configuration.

Dragon also includes soft real-time[6] event delivery architecture. The definition of “Soft Real-Time Event Delivery” based on our awareness of the issues is to notify specific information asynchronously from an object to another within a deadline time in a best effort way. We also provide group framework for distributed services named “composite-service”. Composite-service can define a combination of distributed services, and hide a complicated event-driven system from user. In this paper, we describe the details of design and implementation of Dragon.

The remainder of this paper is organized as follows. In section 2 we describe the core design of Dragon. The fundamental behaviors of Dragon are mentioned in section 3, include a soft real-time event delivering architecture. In section 4, we explain the supporting tools for flexible event management, including composite-service framework. The part of evaluation of Dragon are mentioned in section 4, related works are addressed in section 5, and summary in section 6.

2. Fundamental Elements of Dragon

In this section, we describe the core part and essential components of Dragon.

2.1. Jini and Unicast RMI based System

Before we proceed to the details of Dragon, let us describe some backgrounds and underlying systems. Dragon deals with events among objects. In particular, Dragon adopts an asynchronous event programming model, while other systems may employ polling, callback, or synchronous events. In implementation, an event can consist of serializable Java objects such as string data, integer value, or even mobile codes.

Figure 1 shows the system architecture of Dragon. To provide network address transparency and fault tolerance, Dragon is implemented on Jini. Jini infrastructure is composed of the Lookup Service(LUS) and several protocols. Before any further explanation on Dragon, let us review the process of LUS here. The LUS works as a bridge between services and clients as shown in Figure 2.

1. Once a service has discovered an LUS, it can register itself to the LUS using join protocol, and attempts to maintain itself to LUS using the lease mechanism.

2. A client specifies its requirement using a service template. The service template specifies a type of service required in terms of Java classes/interfaces and a set of attributes.

3. When the client performs a lookup with Lookup protocol, a client obtains a stub object from the LUS.

4. The client can invoke remote methods through well-known service interfaces and registers itself to the service.

5. Then the service notifies events to the clients when something happens.

Figure 2. Jini Distributed Event Architecture

Since Jini is based on RMI, Dragon is under the restriction of RMI. RMI provides mechanisms for unicast communications among different objects by finding, activating, and garbage-collecting the objects. In terms of efficiency, multicast communications would be suitable compared to RMI. However, to retain interoperability with Jini, we adopt RMI for event notification. In addition, since a multicast event delivery using UDP is unreliable, we need another mechanism to ensure the reliability of data. Furthermore handling every multicast event includes useless multicast events. Such reliable control and multicast channel management cost is too much for embedded Java machines.
In order not to limit the variation of application and specification upon the Dragon, we assume that any kinds of contents can be included in an event object, even if java.rmi.MarshalException object, however we are considering that MarshalException object is too unclear and we are researching which kind of event type is suitable for networked appliances and sensors. We do not refer to event contents in this paper.

2.2. Constructor of Basic Service

The combination of services and the members of services change dynamically and quickly. Therefore, we design Dragon as completely push-type event-driven system in Messaging Classification Framework[19] and as distributed non-dominating server system. Basic Service is the most primitive class that have all the elements of distributed components for networked appliances and sensors. Basic Service consists of five software modules: Acting Module, Sensing Module, Event Input Module, Event Output Module, and Event Filtering Module. Programmers can extend Basic Service to any services easily. The functions of these modules are described as follows.

Figure 3. Basic Service components

- **Acting Module (AM)** makes an action against the real world entities, when a service receives an event. It works like a controller of an air conditioner.

- **Sensing Module (SM)** defines how to sense environmental information from real world entities. For example, SM defines the way of obtaining the temperature from the air conditioner. SM triggers event filtering and firing.

- **Event Input Module (EIM)** is an object that handles many kinds of events that have occurred in other services. An EIM receives events from another service and transfer the event objects to Event Filtering Module. EIM is registered in LUS as one of the Jini service, so every service can look up EIM and hold a list of EIM.

- **Event Output Module (EOM)** fires an event to another service in accordance with an ordering table that was created when a user configured the event path. EOM is also registered in LUS as one of the Jini service. EOM maintains the registration list to fire events.

- **Event Filtering Module (EFM)** checks the contents of the events and decides on firing events or not.

Figure 4 shows the constructor of Basic Service. A Basic Service needs four attributes: ServiceMetaInformation, locationGroups, lease_dur_sec, and max_renewal_times.

```
1 public BasicService
2 (Entry[] ServiceMetaInformation,
3 LocationGroups locationGroups,
4 int lease_dur_sec,
5 int max_renewal_times,
6 FilteringModuleInterface filteringmodule,
7 ActingModuleInterface actingmodule,
8 SensingModuleInterface sensingmodule
9 )
```

Figure 4. Constructor of Basic Service

- **ServiceMetaInformation** provides meta information about its own service such as name, vendor name, serial number, and the version number.

- **locationGroups** is information about the LUS that a service is required to join. Both host names of LUS and multicast group's names of LUS can be described in it. Thus, an user can find a service even if whose name is unknown or even if the one which is on a remote host in the Internet.

- **lease_dur_sec** is the duration that the service can take a lease on an entry space for the LUS. If a service changes its IP address frequently, programmers of the services can set this duration to a shorter time.

- **max_renewal_times** is the number of times a service can be leased. A short-lived service must specify a small number of times.

2.3. Event Supplier and Event Consumer

Programmers can create new type of services by extending Basic Service class. Since a Basic Service contains too much information, we divide the function of
Basic Service into three separate services: Event Supplier, Event Consumer, and Event Manager. Programmers can implement a new concrete service based on these services more easily.

This Event Supplier, generating events, is one of a sub-class of Basic Service. Event Supplier needs only a Sensing Module. Unlike Event Manager described below, Event Supplier is suitable to be executed on a machine with low computation power like sensors and mobile devices.

Event Consumer, receiving events, is also a sub-class of Basic Service. The constructor of Event Consumer just needs an Acting Module. This service is also assumed to be run on a machine with low computing power like appliances and mobile devices.

2.4. Event Manager

Event Manager is also a stub-class of Basic Service. This service is the only one that assumed to be running on a high cpu power machine like Set Top Boxes or PCs. In the Filtering Module, a programmer can describe whether or not an incoming event should be passed. The programmer can also describe some operations using the incoming events. This type of service can be called inter-mediator, intermediary or joint-node.

To show the role of Event Manager more concretely, let us assume that there are four thermometers in separated rooms and four Event Suppliers that exist corresponding to each thermometer. In such case, we can calculate the average temperatures using the Filtering Module. It receives four events from the Event Suppliers and computes the average value of four. Another service using the output of this module can control temperature based on the average temperature.

2.5. Distributed Event Registration

A service can be used by other services using registration. An Event Input Module (EIM) of a service registers itself to an Event Output Module (EOM) of another service. Once the registration is completed, the EOM updates its registration list. When an event is generated, it is sent from the EOM to all the registered EIMs. In Figure 5, the EIM of the Event Manager (EM) registers itself to the EOM of the Event Supplier (ES). Similarly, the EIM of the Event Consumer (EC) registers itself to the EOM of the EM. In this way, a path of events from the ES to the EC is created. Dragon provides both the Java API that supports this event registration mechanism and the GUI tool that helps us to set up an event path between services visually.

![Figure 5. Event Propagation for Distributed Services](image)

3. Event Delivery Architecture

In this section we describe the fundamental behaviors of Dragon, especially two types of event delivery mechanisms: normal event delivering and soft real time event delivery.

3.1. Distributed Event Delivering in Order

Figure 6 shows the mechanism about event delivery from one service. An Event Output Module has an Ordering Table (OT) that describes the ordering of stub classes for remote method invocation. In the table of ES, stub objects of EC and EM are listed in the order of registration to ES. In the table of EM, EC's stub objects are listed in the order of registration to EM. Then EM calls EC's notification methods. EC also registers its stub object to ES. Therefore the table of ES holds EC's stub object.

![Figure 6. Distributed Event Delivering Model](image)

If ES senses something to notify in this situation, ES calls each listed stub object's notify method and copies an event object one by one. After EM catches the event object from ES, EM also calls listed remote service's notify method one by one. In other words, EM copies the event object at first to EC, secondly EC, thirdly EC, and finally EC following to the
OT. In this way the event spreads to all the registered services.

3.2. Soft Real-Time Event Delivering

In some systems such as sensor networks in a factory, many events occur simultaneously. In such kind of system, we need to differentiate emergent events from other non-emergent ones; a machine accident needs to be notified faster than other regular events. These are some approaches of hard real-time systems with fixed deadlines, however, we are here at soft real-time system, since we allow heterogeneity in OS and networks.

An event path among services is configured as non-real-time on the Dragon editor. An user can re-register a path to be soft real-time. Figure 7 shows how a soft real-time event is registered. Let us assume that the ES1→EM1→EC2 event path is already created. The user wants to make this path real-time. He needs to re-register the ES1→RT EM1 path as a real-time path. The ES1 creates a new ordering table named Real-Time Ordering Table (RTOT) and moves the EM1's stub from OT to RTOT. After the user changes the EM1→RT EC2 event path real-time, EM1 creates a new RTOT and moves only EC2's stub from a OT to this RTOT. If EM1 receives too many events from ES1 within a short duration under this condition, EM1 skips to copy the event to EC4 of OT and grants that the event object surly copies to EC2 of RTOT with in a dead-line.

![Figure 7. Soft Real-Time Event Registration Scheme](image)

The notifyRT() method at the remote. In the same nature, the service within the OT invokes the notify() method.

![Figure 8. Internal scheme of the Event Manager](image)

Further mechanism of Dragon’s soft real-time architecture, especially dead line time setting way, have been mentioned in [7].

4. Supporting System of Flexible Event Management

In this section, we explain the tools that support flexible event management, include Event Binding Editor.

4.1. Event Binding Editor

Dragon provides a supporting tool name Event Binding Editor that can configure the event paths for distributed services visually.

Figure 9 shows a screenshot of Event Binding Editor. This editor is aimed for supporting users to create ad-hoc event-driven system. Users who have no programming skills can see which type of services exist on the network and can setup the event-driven system that they want to use by using this Event Binding Editor. To enable this editor to be used at the Internet scale as well as at homes, this editor is designed for Jini enable service and automatically detects that a service is launched at the different host. Then, this editor shows the service's icon to user. In this figure, every service is launched at the same host, however, it is common that every service is launched at the different host.

When a user brings a mouse cursor to an icon, the icon shows the service detail information: name, location, vendor name, and serial number. When the user clicks the icon, the icon shows the interface and the line
to connect. Then, the user moves the mouse cursor to another service the icon. If the interfaces match each other, this editor repaints the icon and the line in blue and shows the details of the service. If the interfaces do not match each other, this editor repaints the icons in red. The user clicks blue colored icons to connect to them. This action is transmitted immediately to their target distributed services and each target service tries to register itself to upper service of event stream. In this way, the user can configure event propagation path and can make flexible ad-hoc event-driven system easily. This editor is designed to content synchronized among multiple users in order to ensure system-wide consistency even when the users edit the content simultaneously. For example, a content edited on a PDA is immediately displayed on a screen of a PC.

4.2. Timing Estimation of Event Delivering

Dragon is designed for using at time-restricted situations such as factory work. In such situations, it is necessary to show the user how the time behavior will change when we change the combination of the event-driven system using Event Binding Editor. Users of this editor can notice the event delivery latency and average time of each event path. Each service calculates how much time it took to notify their client. We can confirm millisecond-unit time through a browser. As seen in figure 10, we can identify three types; Average Time(AT), Best Case Time(BT), and Worst Case Time(WT). If the user were to make every event path soft real-time, the browser would show the user not so good AT and WT values as the result.

4.3. Composite-Service Framework

As shown in the illustration of figure 11, the systems that services connected linearly can also be built. In such systems, a mechanism to ensure time-based restriction from the phase when the event is created until the phase that it is fully consumed, will be necessary. This type of problem, whole timing constrain of distributed components, is not supported now, however, we aim to solve this problem as one of the future work.

![Figure 10. A screen shot of event timing estimation phase](image)

![Figure 11. A screen shot when user sets up event path linearly](image)
composite-service has ports to fire events named Ex-
ternal Event Output Module but no ports for events 
reception. On the other hand, supplier type composite-
service has ports for events reception named Ex-
ternal Event Input Module but no ports to fire events. 
Intermediate type composite-service has both Ex-
ternal Event Output Module and External Event Output 
Module. Every composite-service can permit events to 
come from only particular ports and can limit events to 
fire from particular ports as shown in figure 13. Users 
of this system can make a composite-service locked via 
Event Binding Editor and reduce the miss-operation of 
the system.

Figure 12. A screen shot where the user defines 
composite-service using Event Binding Editor

Figure 13. Dragon limits each event to 
come in composite-service from only the 
particular port

5. Related Work

In this section, we describe some related works to 
flexible event notification and event delivery models 
with time constraint.

A Reflection Based Tool for Observing Jini service 
(Carp®)[5] is the management tool for dynamic dis-
tributed Jini systems. Carp® allows observation of ser-
ices and clients, their interconnections and the mes-
gages exchanged among them. The tool extracts an 
architectural component model based on components, 
ports and connectors. However, Carpat is not aimed 
at time restriction unlike Dragon. Furthermore, we 
focus on the event propagation path by connecting Event 
Managers linearly.

Agent Anytime Anyware[9] also provides both the 
configuration graphic tool and notification-based mech-
anism. C2 [13] provides a GUI configuration tool based 
on JavaBeans named “arilica”. To compare with Drag-
on, these above two research have richer function of 
GUI tools. For example property configuration dialog 
and saving mechanism. However, Dragon’s GUI sup-
porting tools dynamically detects service birth, expiry, 
and attribute changing, because we assume that Drag-
on is used at home and factory.

Event Centric for Java (ECJ)[3] is an event delivery 
framework for developing Java applications distributed 
over networks. ECJ introduces a network transparent 
event-driven framework into Java so as to integrate 
object oriented systems with event driven ones. ECJ 
has an asynchronous message exchange infrastructure. 
However, ECJ does not support discovery services the 
location is unknown and can not change is event noti-
fication target dynamically. Further more, ECJ does 
not have the end-user friendly software like the Dragon 
editor.

Java Event Channel (JEcho)[21] aims to support 
distributed group communication by offering the not-
ions of events and event channels. Event Channel is 
a logical construction that links some number of end-
points to each other like Dragon’s composite-service. 
JEcho employs multicast transmission. Therefore, ev-
ery distributed object must support and understand 
related and non-related event. In the case of low com-
putation power machine like sensors and appliances, 
to control multicast channel and groups is a difficult 
problem. To manage which service has to join which 
multicast channel and how to discover the services the 
user wants, is also a difficult problem.

SIENA[2] is middleware for distributed event ser-
vice. SIENA restricts type of event’s contents strict-
ly, because SIENA is depend on contents based event 
routing. SIENA also offers “patterns”. “Pattern” is 
an expression whose basic elements are filters. How-
ever, this patterns have very restricted expression. On 
the contrary, Dragon permits any type of serializable 
object as event object even if java RMI stub, so pro-
grammers of each Dragon services have to decide the 
detailed behaviors of each Event Filtering Module. We 
can adapt more strict filtering models, including SIEN-
A’s Pattern model. This is one of the future topics of
this research.

6. Summary and Future Work

In this paper, we presented a new framework for constructing flexible inter-service event-driven systems, referred to as Dragon. Dragon includes a soft real-time event delivering architecture and an composite-service framework. Dragon support service to notify specific information asynchronously from an object to another within a deadline time in a best effort way. This system provides three benefits. First, programming for distributed services like networked appliances can reduce the programming cost by using Dragon’s typical service classes: Event Supplier, Event Manager, and Event Consumer. Each service class has not so rich functions and powerful mechanism, however the combination of these service class enable any programmers and end-users to create any type of distributed sensors and appliances system. We can bind some helpful typical combination for services called “Composite-Service”, then programmers can hide complex combination of services from users of this system. Second, the users can create ah-hoc distributed systems without cost of configuration by using GUI named Event Binding Editor. This editor supports that users can control whole the disturbed event-driven system without programming intuitively. Third, the system that the users creates becomes more time reliable by using soft real-time event delivery architecture of Dragon.

We have four major future works: the use of RT-JAVA on RT-Mach[20], adaptation for real-time streaming data[14], content level event path registration, prediction of event dead lock.

Acknowledgment

The authors are grateful to members of the Java Project for valuable comments and criticisms, especially Yoshiho Tobe, Nobuhiko Nishio of Keio University, and Hidehiko Wada of Yokogawa Electric Corporation supplied ideas for extending this research. We thank anonymous DOE 91 reviewers for several comments and suggestions that helped improve the quality of this paper.

References