IEEE The Seventh International Conference on Real-Time Systems and Applications 2000 (IEEE RTCSA’00) @Korea, Cheju Dec.’00 Proceedings pp425-pp432

**Dragon: Soft Real-Time Event Delivering Architecture for Networked Sensors and Appliances**

Masayuki Iwai\(^1\), Jin Nakazawa\(^1\), and Hideyuki Tokuda\(^2\)\(^1\)
\(^1\)Graduate School of Media and Governance, \(^2\)Faculty of Environmental Information, Keio University
5322 Endo, Fujisawa, Kanagawa 252-8520, Japan
{tailor, jin, hxt}@ht.sfc.keio.ac.jp

**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 top of 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. In this paper, we describe in detail 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. As one aspect, network-ready hardware such as TINI\(^2\) have accelerated this trend. Software for embedded devices such as light weight Java\(^2\) and real-time Java\(^2\), facilitate combinations of the software that is distributed over a network. Furthermore, middleware such as HAri\(^2\), UPnP\(^2\), VNA\(^2\), and Jini\(^2\) provide plug-and-play facilite. 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. First, services, components software that constitutes appliances and sensors may change. They may 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 happens. 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, real-time 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. 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 re-configurable 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.

Dragon reduces both the programmer’s developing cost and end-user’s combinational configuration cost by supporting a graphical editor of combining events between services. Dragon is aimed 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 Java Beans Editor[7], it differs in that the actual constructed system is reconfigured immediately after it is edited on the Dragon editor and in that the Dragon is aimed for distributed services configuration.

Dragon also includes soft real-time[7] event delivering 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. 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, and in section 3 we explain a real-time event delivering architecture. The evaluation and an application of Dragon are mentioned in section 4, related works are addressed in section 5, and summary in section 6.

2. Distributed Event Delivering Framework

In this section, we describe the core part and 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 be consisted 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.

Figure 2. Jini Distributed Event Architecture

Since Jini is based on RMI, Dragon is under the restriction of RMI. RMI provides the mechanisms for unicast communications among different objects by finding, activating, and garbage-collecting the objects. By efficiency, MulticastSocket 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 if we adopt the MulticastSocket.

2.2. Construct of Basic Service

A Basic Service is the most primitive class in Dragon. A Basic Service consists of five software modules: Acting Module, Sensing Module, Event Input Module, Event Output Module, and Event Filtering Module. The functions of these modules are described as follows.

- 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 transfers the event objects to Event Filtering Module. $EIM$ is registered in LUS as one of the Jini service, so we can look up $EIM$ and we can 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 registered in LUS as one of the Jini service, so we can look up $EOM$ and we can register the list of $EIM$ to $EOM$. $EOM$ 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 3. Basic Service components

Figure 3. Basic Service components

A Basic Service also needs four attributes: *Service Meta Information*, *locationGroups*, *lease_dur_sec*, and *max_renewal_times*.

```
1 class BasicService
2 2 (Entry[] ServiceMetaInformation, 
3        LocationGroups locationGroups, 
4        int lease_dur_sec, 
5        int max_renewal_times, 
6        FilteringModule filteringmodule, 
7        ActingModule actingmodule, 
8        SensingModule sensingmodule 
9 )
```

Figure 4. Constructor of Basic Service

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

- **locationGroups** informs the LUS that a service is required to join. Both host names and Jini LUS group's names of LUS can be described in it. Thus, an end-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 can set this duration to a shorter time.

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

### 2.3 Event Supplier and Event Consumer

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. Figure 5 shows a constructor of Event Supplier. 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. A class that corresponds to a sensor will be implemented with Event Supplier.

```
1 class EventSupplier extends BasicService{
2    public EventSupplier
3    (Entry[] ServiceMetaInformation, 
4        LocationGroups locationGroups, 
5        int lease_dur_sec, 
6        int max_renewal_times, 
7        SensingModule sensingmodule){
8        super(ServiceMetaInformation, 
9        locationGroups, 
10        lease_dur_sec, 
11        max_renewal_times, 
12        null, //FilteringModule 
13        null, //ActingModule 
14        sensingmodule); 
15    }
16 }
```

Figure 5. Constructor of Event Supplier

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

2.4. Event Manager

Event Manager is also a stub-class of Basic Service. Figure 7 shows a constructor of Event Manager. 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.

```java
class EventManager extends BasicService{
public EventManager
  Entry[] ServiceMetaInformation,
  LocationGroups locationGroups,
  int lease_dur_sec,
  int max_renewal_times,
  FilteringModule filteringmodule){
  super(ServiceMetaInformation,
    locationGroups,
    lease_dur_sec,
    max_renewal_times,
    filteringmodule,
    null//ActingModule
  null//SensingModule);
}
}
```

**Figure 6. Constructor of Event Manager**

Let us assume that there are four thermometers in separated rooms and four Event Suppliers that exist corresponding to each temperature. 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. This service should be executed on a high power machine.

There are two special sub-classes of Event Manager: Event Merging Manager and Event Timing Manager. Both of them do not depend on the contents of incoming events.

**Event Merging Manager** awaits all the events coming from several services and merges these events together for firing an event to the service that is registered.

**Event Timing Manager** holds the events to fire until the time that an end-user set. For example, the end-user can set this service to send events every half hour.

2.5. Distributed Event Registration / Firing Architecture

A service can be used by other services using registration. An EIM of a service registers itself to an 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 7, the EIM of the EM registers itself to the EOM of the ES. Similarly, the EIM of the EC registers itself to the EOM of the EM. In this way, a path of events from the ES to the EC is created.

![Event Propagation for Distributed Services](image)

**Figure 7. Event Propagation for Distributed Services**

2.6. Event Binding Editor

Figure 7 shows a screenshot of Event Binding Editor. Event Binding Editor provides the end-user with ServiceMetaInformation, mentioned before, of all the services, thereby enabling the end-user to configure the binding of services. To enable this editor to be used at the Internet scale as well as at homes, it is divided into three modules: Event Binding Proxy (EBP), Event Binding Applet (EBA), and Event Binding Applet Client (EBAC).

The end-user from another multicast segment using an EBAC asks the remote LUS for the location of an EBP. While the EBAC finds the EBP, the EBAC tries to download an EBA from the EBP. Then the end-user can use this editor. The EBP searches all the LUSes corresponding to a locationGroups. When a service is destroyed and cannot lease itself to an LUS, the LUS notifies the editor that the service becomes unavailable. Upon notification, icon of the service becomes dim.

When the end-user brings the mouse cursor to an icon, the icon shows the service detail information: name, location, vendor name, and serial number. When the end-user clicks icon, the icon shows the interface and the line to connect. Then, the end-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 does not match each other, this editor repaints the icons in red. The end-user clicks blue colored icons to connect to each other. 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 end-user can configure event propagation and can make ad-hoc distributed system easily.

![Figure 8. A screen snapshot of Event Binding Editor](image)

### 2.7. Distributed Event Delivering Order

An Event Output Module of every services has an ordering table named Non-Real-Time Ordering Table (NRTOT) that describes the ordering of stub classes for remote method invocation. Figure 7 shows the mechanism about the method invocation order. In the table of $ES_1$, stub objects of $EC_5$ and $EM_1$ are listed in the order of registration to $ES_1$. Then $ES_1$ calls $EM_1$’s and $EC_5$’s notification methods. In the table of $EM_1$, $EC$’s stub objects are listed in the order of registration to $EM_1$. Then $EM_1$ calls $EC$’s notification methods. $EC_5$ also registers its stub object to $ES_2$. Therefore the table of $ES_2$ holds $EC_5$’s stub object. Then notification methods of each stub objects are called one by one. On this way if $ES_1$ senses something to notify, the events spread to $EM_1$ and $EC$s.

![Figure 9. Distributed Event Delivering Model](image)

### 3. Soft Real-Time Event Delivering Architecture

In this section, we describe Soft Real-Time Event Delivering architecture in Dragon.

### 3.1. Scenarios

In some stem such as sensor networks in a factory, many events occur simultaneously. Event in these systems, we need to differentiate emergent events from other non-emergent ones; a machine accident needs to be notified faster than other regular events. There are approaches of hard real-time systems with firm deadlines. We here aim at soft real-time system, since we allow heterogeneousness in OS and networks.

### 3.2. Notation

We define the notation of this system. $ES_1 \rightarrow EC_1$ means that the Event Supplier $ES_1$ fires events to the Event Consumer $EC_1$ without Soft Real-time Event Delivery Architecture. $ES_1 \rightarrow EM_1 \rightarrow EM_2 \rightarrow EM_3 \rightarrow EM_4 \rightarrow EC_1$ means that the event generated by $ES_1$ was carried to $EC_1$ via $EM_1$, $EM_2$, $EM_3$, and $EM_4$. $ES_1 \rightarrow \{RTEC_1, EC_2, EC_3\}$ shows that $ES_1$ has three clients to notify the events. Especially $EC_1$ is the only client that have to notify the event with Soft Real Time Event Delivery.

### 3.3. Real-Time Event Registration

An event path among services is configured as non-real-time on the Dragon editor. An end-user can re-register a path to be real-time. Figure 7 shows how a real-time event is registered. Let us assume that the $ES_1 \rightarrow EM_1 \rightarrow EC_2$ event path is already created. The end-user wants to make this path real-time. He needs to re-register the $ES_1 \rightarrow RT EM_1$ path as a real-time path. The $ES_1$ creates a new ordering table named Real-Time Ordering Table (RTOT) and moves the $ES_1$’s stub from $NRTOT$ to $RTOT$. After the end-user changes the $EM_1 \rightarrow RT EC_2$ event path real-time, $EM_1$ creates a new $RTOT$ and moves only $EC_2$’s stub from a $NRTOT$ to this $RTOT$. 
3.4. Differentiation of Real-Time Event and Non-Real-Time Event

In RTOT, the stub object that has earliest deadline are listed in higher position as shown in figure 77. Java VM supports three static priorities of threads: MAX_PRIORITY, MIN_PRIORITY, and NORMAL_PRIORITY. Two threads deal with scheduling of events in EOM. Real-Time Event Handler (RTEH) thread runs with the MAX_PRIORITY and schedules real-time events. The other thread, Non-Real-Time Event Handler (NRTEH) thread, runs with the MIN_PRIORITY and schedules non-real-time events. NRTEH can only be executed when there are no real-time events.

3.5. Analysis of Event Arrival Time

Dragon also offers the timing analysis of the Soft Real-Time events notification. We explain the sequence of timing analysis. After an end-user configures the real-time system using Event Binding Editor (EB), EB commands register EC2 to the EM1. EM1 attempts to invoke a test method at EC2 ten times and measure the average time for remote method invocation. EM1 informs the average time, the worst-case, and the best-case time to EB1, and EB1 shows the average time to the end-user immediately. If end-user sets a deadline of the communication between EM1 and EC2, EM1 also analyzes the time and calculates success ratio that indicates how many of the specified real-time events satisfies the deadline.

Timing analysis measures the time for the duration from the time that a service begins to create the RTEH until it finishes a remote method invocation to one’s clients more than ten times. After this measurement, the service calculates the average, the worst case, and the best case the time and success ratio of inter-services communication and notifies Event Binding Editor of the calculated result.

Figure 77 is the screen snapshot that end-user configures the real-time event bindings. The user can confirm three values. Average time (AT) is the average of the event notification test. Deadline (DL) is the time set by the user using Event Binding Editor.

If the same values of the DL are set to one service, the first one acquires a higher ordering priority than the second. Success Ratio (SR) is the ratio of how many times the event notification is done within DL. The end-user confirms these values and designs an ad-hoc system with time reliability.

4. Evaluation and Application

This section analyzes the performance of Dragon and describes one application to explain Dragon is effective in some environment.

4.1. Evaluation

Figure 77 shows the duration time to deliver event to all the clients. In a case where a service has many clients Non real time event delivery takes much time because of using unicast. The more numbers of clients to register a service, the duration time rises in proportion to a rise in the number of non real time clients. If we do not consider soft event delivery, we cannot figure out the event delivery cost on the way to clients.

At one service that have both one real time client and several non clients, we evaluated the duration average time to finish notifying all the clients and the duration average time to finish notifying one real time
According to figure 77, no matter how many non real time clients registers to a service, one real time event registration keeps a quite stable duration time.

Even in such small room space Dragon displays its ability.

5. Related Work

In this section, we describe some related works to event notification model and real-time event delivery model.

CORBA Event Service is one of the Common Object Services in CORBA specifications. CORBA Event Service defines a set of interfaces that provide the synchronous event communication mechanism. The interfaces support a pull-style and a push-style communication. Furthermore, multicast distribution among event suppliers and event consumers is accommodated with event channels. Dragon has just event driven (push-style) communication, however Dragon can communicate with pull-style using Event Manager. Unlike event data of CORBA Event Service, Dragon on Jini structure can notify event as an object.

The Real-Time CORBA[27] is the scheme of the scheduling of event groups on one ORB object. Dragon does not decide one object for event controlling. All the services have an soft real-time event delivery mechanism. Our architecture is superior to real-time event notification distributed and heterogeneous in the situations.

A Reflection Based Tool for Observing Jini service (Carp®)[32] is the management tool for dynamic distributed Jini systems. Carp® allows observation of services and clients, their interconnections and the messages exchanged among them. The tool extracts an architectural component model based on components, ports and connectors. Since the Dragon editor is an Java applet, it is controlled more flexibly through www browser. Dragon can offer a more flexible event communication using Event Manager than Carp®. In addition, Dragon can offer Soft Real-Time event delivery.

Figure 13. Non real time event delivery total duration time

Figure 14. \( ES_1 \rightarrow \{RT, EC_1, EC_2, ..., EC_{n-1}\} \) system

Figure 15. Embedded appliances and devices are inside Smart Space Lab
Event Centric for Java (ECJ)\cite{1} 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 event notification target dynamically. Further more, ECJ does not have the end-user friendly software like the Dragon editor.

Java Event Channel (JECho)\cite{2} aims to support distributed group communication by offering the notions of events and event channels. Event channel is a logical construction that links some number of endpoints to each other. JECho employs multicast transmission. Therefore, every distributed object must support and understand related and non-related event. In the case of low computation 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.

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. 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. Second, the end-user can create ah-hoc distributed systems without cost of configuration by using the Event Binding Editor. Third, the system that the end-user 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\cite{3}, adaptation for real-time streaming data\cite{4}, the CORBA3.0 interoperability, aggregation of services, and saving mechanism of the ad-hoc system. Especially, using RT-Java we want to make dragon to be more hard real-time based system. Several priority threads running in the Dragon service, events arrive asynchronously and priority inversion problems occur. Dragon’s latest software version, manuals, and evaluations are available at our web site http://www.ht.sfc.keio.ac.jp/~tailor/dragon/

Acknowledgment

The authors are grateful to members of the RT-HDI Project for valuable comments and criticisms, especially Yoshito Tobe, Nobuhiko Nishio of Keio University, and Hischiko Wada of Yokogawa Electric Corporation supplied ideas for extending this research. We thank anonymous RTCSA’00 reviewers for several comments and suggestions that helped improve the quality of this paper.

References


