Implementation of a Distributed Network Middleware “CSC” on OSGi Frameworks

Ikuo Yamasaki, Kouji Yata, Hiroyuki Maeomichi, Akihiro Tsutsui, and Ryutaro Kawamura
Cyber Solutions Laboratories
NTT Corporation
Yokosuka, Japan
{yamasaki.ikuo, yata.kouji, maeomichi.hiroyuki, tsutsui.akihiro, kawamura.ryutaro}@lab.ntt.co.jp

Abstract—To make communication a comfortable experience for users, we have developed a middleware called Communication Service Concierge (CSC). This is a network middleware for distributed environments, and it consists of Communication Resource Management (CRM) and Communication Logic Management (CLM). Communication Service Concierge realizes comfortable communications by combining CRM and CLM. Communication Logic Management provides a framework to directly operate communication data. Communication Resource Management provides a framework on which downloadable software components can be executed on distributed platforms, and plays a primary role in controlling CSC’s communication. Using Java, we implemented CRM on OSGi service platforms, which provide dynamic executions of the software components locally. In other words, CRM provides a distributed agent computing feature for OSGi. In this paper we describe the roles and implementations of CRM, especially for security functions, as well as describing how CRM’s software components work cooperatively.

Keywords—CSC, Network Middleware, Java, OSGI

I. INTRODUCTION

The Internet has progressed rapidly in the last few years, especially in current access technologies, such as ADSL and FTTH that provide broadband and 24-hour Internet connections. The Internet is rapidly becoming an indispensable commodity for people. This remarkable Internet growth has yielded many new network technologies and services in recent years. These technologies and services are evolving more rapidly than ever before and are diversifying, not only to various network applications that include e-mail and web browsing, but also to video streaming and P2P file sharing. In addition, the users’ objectives and communication requirements are also becoming more diversified.

When using communication applications, users are forced to make many choices to achieve ‘comfortable communication’. Moreover, because of the diversity and complexity of the net and its applications, problems that prevent ‘comfortable communication’ are scattered throughout the end-to-end communication link. Advances to solve each problem might be available including QoS management technology, such as DiffServ [1], which controls the network’s QoS, UPnP [2] and Jini [3], which facilitate the automatic configuration of home devices and load sharing technology, used to relieve server bottlenecks. Thus far, these technologies have been developed in a piecemeal manner by different industry players, such as communication network operators, operating system (OS) providers, and telecommunication equipment vendors. They are relevant only to specific portions of end-to-end communication links.

To address this problem, we have developed a new type of communication middleware called the Communication Service Concierge (CSC) [4], [5], and [6], that provides a communication control framework, based on distributed agent computing. It performs end-to-end seamless communication control to achieve ‘comfortable communication’ that satisfies network users in the areas of communication usability, availability, quality, and safety. CSC consists of Communication Resource Management (CRM) and Communication Logic Management (CLM). While CLM provides the framework to directly operate a communication flow, CRM provides the framework where downloadable software components can be executed on distributed platforms, and plays a primary role in CSC’s communication control. We implemented CRM on the Open Services Gateway initiative (OSGi) frameworks, which provide dynamic executions of software components locally. In other words, CRM provides a distributed agent-computing feature for OSGi frameworks.

The rest of this paper is organized as follows: Section II briefly introduces CSC. In Section III, we show the roles of CRM and relationship between CRM and OSGi. Section IV specifies the CRM security functions and our implementations. In section V, we describe the ways that CRM offers for software modules to cooperate with each other. In section VI, we conclude our paper.

II. NETWORK MIDDLEWARE CSC

A. Design Concept

The Internet includes a diverse range of network technologies and services. To make communication via the Internet a more comfortable experience, we require seamless communication control that encompasses the entire end-to-end communication links. However, we do not consider that achieving ‘comfortable communication’ means only to guarantee network QoS requirements, such as bandwidth, latency, and jitter. We also believe that it includes the following elements.
Easy to start, easy to use: The user should be able to communicate easily, without requiring expert knowledge and complicated settings.

Stress-free: It should provide the network QoS attributes (bandwidth, latency, jitter, etc.), required by the user and the application.

Ubiquitousness: Communication should be available anywhere at any time.

Safety and Security: Users should always be able to conduct secure communication. Moreover, they should be able to enjoy communication services that will satisfy their requirements at a reasonable cost.

Flexibility and configurability are the primary CSC design concepts, because it needs to control a wide variety of objects. Moreover, CSC is designed not to tightly link network technologies and applications, in order to sustain their independence. To enable CSC to support various execution platforms, it is implemented using Java, which has a low operating system dependency. In addition, we had to take the small implementation footprints and the ability to run on mobile devices and home network appliances into consideration.

B. Control Model

Figure 1 shows the CSC control model that focuses on the communication flow while considering the hardware and software elements related to the flow as CRs that build the communication. The term CR refers to the communication resource. In this model, the CRs include a wide variety of elements, such as communication terminals (PCs, STBs, etc.), network nodes (routers, gateways, etc.), application software, and users in communication sessions. The communication flow behavior depends on the combination, configuration, and status of these CRs. The CSC coordinates these CRs to achieve comfortable communication.

To coordinate and control the various CRs, scattered throughout the end-to-end communication links, CSC deploys software components that monitor or control the CRs. These components are called plug-in modules (PIs). Each PI might have an access method and/or a control algorithm for a target CR. For example, a PI that accesses a router and configures its packet scheduling mechanism, includes an access protocol for the router (SNMP, HTTP, TELNET, etc.), and/or a packet prioritization algorithm. The CSC provides a framework that manages the PIs and allocates them to appropriate locations. The PIs are stored in PI servers, located on the Internet, and are downloaded dynamically, as needed. A single PI can control the other PIs in a hierarchical fashion. The PI that has the highest level of communication control is called the ‘root module’, and it is usually loaded first. Thereafter, the PIs best suited to the control of specific CRs are downloaded. The root module may receive user requirements through its Graphical User Interface.

C. CSC Architecture

Fig. 2 shows the CSC architecture reference model. CSC consists of two control frameworks called Communication Resource Management (CRM) and Communication Logic Management (CLM). The basic concept of CSC is realized by CRM, which plays a primary role in CSC’s communication control, and is a distributed communication control framework that coordinates and controls CRs through plug-in based architecture. However, CRM cannot handle communication flow data directly because it controls the flow indirectly through the CRs. On the other hand, CLM is a control framework that can operate a communication flow directly, and it is located between the network application and the operating system. Plug-in modules of CLM are inserted into the communication flow, and they execute direct data-flow operations, such as transport protocol, CODEC, and encryption. This CLM framework can be regarded as a ‘functional socket’, with a dynamic protocol stack or filter.

Both CRM and CLM consist of core parts (CRM and CLM cores) and plug-in modules (CRMMs and CLMMs). The core provides a plug-in module-executing environment, and is in devices that run CSC. (If only one CRM or CLM is to run in a device, only the corresponding core part is required.) Some examples of the functions performed by the core parts include downloading, starting the plug-in modules, and security functions. Moreover, the core parts themselves have no communication control algorithms. They are designed to minimize their footprints and required processing loads.

On the other hand, plug-in modules have communication control algorithms and the ability to access CRs. They are downloaded and executed by the core part from a plug-in module server (or from a local storage device).
The CRM cores and CRMMs are installed on multiple devices and inter-work with each other as a distributed system to provide communication control. CRM can run on any device. (It does not need to be on the computer that contains a CR.) On the other hand, CLM can run in an executing environment, confined to a single device. In other words, a CLM core does not inter-work (communicate) with other CLM cores. However, CLMMs may communicate with each other if necessary, using a proprietary communication scheme.

A plug-in module server stores CRMMs and CLMMs. In CSCs, the plug-in modules required for a particular application can be dynamically downloaded and executed from the plug-in module server. Of course, we can also pre-install the required CRMMs and CLMMs in devices.

Since we discussed CLMs in detail, along with examples using CRM and CLM, in previous papers [4], [5], and [6], we will focus mainly on the CRM implementation details in this paper.

III. CRM

A. CRM and Open Service Gateway initiative (OSGi) Service Platform

One of the most fundamental roles of the CRM core is to provide a distributed object environment where CRMMs can be executed dynamically. That is, a CRMM can download and execute other CRMMs dynamically at appropriate CRM core locations (the device on which the CRM core is running), either locally or remotely.

Currently we can realize functions, such as downloading, installing, executing, stopping, and removing CRMMs, on the platform, using OSGi framework [8], [9], [10], and [11]. In OSGi terminology, a plug-in module is called a ‘bundle’. Bundles are executed on the OSGi framework, which provides life cycle management of bundles only locally, not remotely. In the rest of the paper, “FW” and “core” represent the OSGi framework and CRM core, respectively. On a FW, bundles can share the Java package, and the “service” which a bundle provides can be registered dynamically to the FW, without rebooting the Java virtual machines (VM) and FW. Additionally, other bundles can use the service.

We implemented each CRM core and CRMM as a ‘bundle’. Therefore, CRM provides the distributed agent-computing feature for OSGi middleware.

B. CRM Core Service Interface

A CRM core registers the “CRM Core service” to its FW when the core starts. By calling the service, CRMMs can achieve the following:

- **Request to activate CRMM:** CRMMs can request the downloading and execution of the desired CRMM at an appropriate core location, either locally or remotely. The command name is “executeModule()”; whose arguments can specify the core ID, CRMM URL, the CRMM lifetime, and so on. Each core has a unique ID in the Internet, one that is associated with the IP address.

- **Managing the CRMM tree:** If the CRMMs are executed at the request of other CRMMs via cores, the control tree can be built up in a hierarchical manner. Using the core IDs enables the CRMMs to manage the whole structure of the control tree. Once the tree is built using the “executeModule()”, CRMM can offer other commands remotely, via the cores. For instance, CRMMs can stop and uninstall the desired CRMM by using the “eliminateModule()” command along the tree.

- **Managing CRMM Tree Lifecycles:** When the lifetime specified by the “executeModule()” expires, or an “eliminateModule()” command is received, the core will eliminate it, and the tree will be rebuilt. For instance, when the CRMM in the parent position is eliminated, a CRMM in the child position, which doesn’t have other parents, will automatically be eliminated by the core. That is, the core communicates with the other cores to manage the CRMMs’ lifetime, running both locally and remotely.

The CRM establishes an agent-based distributed computing environment, where various types of CRMMs can be executed according to the request received from other CRMMs via other cores. Therefore, one of the most important issues for realizing CRM, is system security.

During activation, the CRMMs might need to communicate and cooperate with other CRMMs on either a local or remote core. How CRMMs can cooperate with each other is important for CRM.

Section IV and V will show how the security functions and the CRMM’s methods of cooperation are realized in CRM.

IV. CRM SECURITY FUNCTIONS

A. Definitions of CRM Security Functions

As mentioned, system security is one of the most important issues for realizing CRM. Fig. 3 shows the CRM security functions that we have defined. They consist of three functions;

- **Sec-F1:** mutual authentication between cores,
- **Sec-F2:** pre-execution CRMM check, and
- **Sec-F3:** CRMM’s resource access control.

![Figure 3. CRM Security functions.](image-url)
1) Security Function 1 (Sec-F1): mutual authentication between cores

Cores communicate with each other in place of CRMMs for some operations, e.g. “executeModule()”, “eliminateModule()”, and so on. Operations between cores are permitted, subject to a ‘contract’ between the two concerned cores. Each core must confirm that the corresponding core is trusted.

To prevent malicious operations by impostors, Sec-F1 should be able to:

- Mutually authenticate between cores:
  - Accurately identify the counterpart core.
  - Decide about whether to receive requests from the identified core, subject to acceptance policies.
- Prevent tampering with communication data and eavesdropping.

2) Security Function 2 (Sec-F2): pre-execution CRMM check

When a new CRMM is to be activated, it is downloaded from and executed by a plug-in module server. Usually, both the CRMM developer and the plug-in module server owner are different from the owner of the core that has issued the ‘activate’ command to the new CRMM. To prevent the activation of malicious CRMMs, Sec-F2 should be able to:

- Admission control
  - Identify CRMM developers
  - Decide about whether to execute, based on its URL and the identification results, subject to acceptance policies.
- Detect CRMM tampering.

3) Security Function 3 (Sec-F3): CRMM’s resource access control

An activated CRMM may need to access the local computing resources on which the core is running. It may also need to access the network. To protect the system from a CRMM maliciously or incorrectly accessing its resources, Sec-F3 should be able to:

- Restrict the resources that a CRMM is allowed to access:
  - Precisely define access restrictions, based on the URL and the developers, subject to access policies.
  - Control resource access during runtime.

In the next section, we will describe in detail how these security functions are realized and our current CRM implementation works on FWs.

B. Implementations of Security Functions

1) Implementation of Sec-F1

We realized Sec F1 using a public key infrastructure (PKI) and secure socket layer (SSL vers. 3.0). Each core selects its own public and secret key for its digital signature in advance. The principal (owner) of the secret key is assumed to be the core owner. In addition, core has a trust-store, which stores the trust certificates of others.

When booting, the core reads its secret key. When the core receives a command to activate a new module on itself on the remote cores from a CRMM, an SSL session between the cores tries to establish itself, where both server authentication and client authentication are performed. In this SSL handshake, each core identifies the subject (owner) of the certificate, based on the trust-store identification system (Core Owner Identification) and decides whether to admit the SSL session, based on the core information about the identified owner and its admission policy (SSL admission control).

2) Implementation of Sec-F2

Sec-F2 is realized using PKI. Sec-F2 consists of “tampering verification”, “signer identification” and “execution admission control”. Before installing the target CRMM, the following steps are completed. First, the core downloads temporarily and verifies whether it is being tampered with or not, using digital signatures (tampering verification). Next, it identifies the signers (signer identification). Finally, using the URL and the identification results, the core determines whether to admit and execute the CRMM, based on policy (execution admission control).

3) Implementation of Sec-F3

In Sec-F3, the current OSGi framework (OSGi Service Platform Release 3) provides a function, but also causes a limitation. Prior to the explanation of our Sec-F3 implementation, we will describe this limitation.

Ordinarily, with Java2 security, the runtime access control of executable Java codes can be achieved, based on information from the digital signature and codebase (URL). This satisfies the requirement for Sec-F3. On the other hand, an OSGi FW provides a “PermissionAdmin” service, which is the only way to dynamically set permissions against each bundle on the FW [11]. However, using this service, the target bundle can be specified, based only on the codebase, which is known as the “bundle location” in OSGi terminology. Therefore, on OSGi FWs, the information that can be referred for access control at run-time, is not digital signatures but only a “bundle location”. We think this is not secure enough for Sec-F3.

We realized Sec-F3, by combining it with Sec-F2. Sec-F3 consists of “permission determination”, “set permissions” and “CRMM execution”. If admitted to ‘execute’ in Sec-F2, the identification results of the signatures in Sec-F2 will be handed to “permission determination” in Sec-F3. Permission determination decides which permissions to set against the CRMM, based on both the signers and its URL. Then, Sec-F3 sets the permissions to the CRMM using PermissionAdmin service (Set Permissions). Finally, after installing and starting the CRMM as an OSGi bundle, the core executes CRMM through the CRM core service (CRMM execution). While it is activated, Java security can do runtime permission checks on OSGi FW.

4) Separation of Policy Implementors and CRM Core

In Sec-F1’s core owner identification and SSL admission control, Sec-F2’s signer identification and execution admission control, and Sec-F3’s permission determination, there might be
a lot of policy implementations, in terms of digital signer identification, decision algorithms, and policy database management.

Therefore, we defined an OSGi service called “CRM security policy manager (CSPM)”; CSPM plays a role in all of them. We implemented the default CSPM, which is included in the core bundle. On the other hand, we let programmers implement their own original CSPMs as OSGi bundles, and use them instead of the default CSPMs. We will now describe how core and CSPM works.

If the default CSPM is used, each core will register the CSPM service to its FW when booting. Otherwise, the core will install and start the original CSPM bundle and register its service.

To easily understand this, we will consider a case where CRMM1 on core 1 asks core 2 to download CRMM2 from URL2 and execute it on core 2, using the “executeModule” command. Fig. 4 shows the sequence of this example.

In the SSL session handshake between the cores, each CSPM provides core owner identification and SSL admission control.

After tampering verification in Sec-F2, core 2 calls a method of CSPM2 service, with the arguments (1) the certificates embedded in the CRMM2 digital signatures, (2) the CRMM URL to execute, URL2, and (3) information about the requesting CRMM. The third argument includes core 1’s ID, the owner’s name, and CRMM1’s URL. These information can be used for execution admission control and permission determination.

After permission determination, CSPM returns the runtime permissions to set the specified CRMM2. Finally core 2 configures the permissions to URL 2 through the PermissionAdmin service, and CRMM2 is installed and executed.

V. COOPERATION BETWEEN CRMMs

While current CRM specifications do not restrict how CRMMs communicate and cooperate with each other, CRM provides two methods of cooperation between CRMMs. The first one uses a CRM core service, “sendMessage()” command. After the CRMM trees have been built, the CRMMs can cooperate with other CRMMs, even on local or remote cores by sending string messages through “sendMessage()” (Fig. 5). The second one uses OSGi services, which are not CRM core services but original implementations. The details of the latter will be shown below.

Since a CRMM is a bundle on the FW, one CRMM can provide some originally implemented OSGi services that other CRMMs can use on the local FW. Normally, CRMMs cannot use services that are provided by other CRMMs on remote cores (FWs). If the CRMM2Server on core 2 provides an originally implemented OSGi service “servicea”, CRMM2Client on core 2 can use “servicea”, but CRMM1Client on core 1 cannot use the service.

We propose a method that enables the CRMMs to cooperate with other CRMMs on remote cores through OSGi services: CRMM1Client on core 1 (FW1) can use the “servicea” that CRMM2Server provides on core 2 (FW2). To realize this, we used an RMI proxies approach. It requires two proxy CRMMs,
an RMI client proxy CRMM (RmiClientM) on core 1 (FW1), and an RMI server proxy CRMM (RmiServerM) on core 2 (FW2), where the RmiClientM connects to the RmiServerM via RMI (Fig. 6). In this approach, when a CRMMClient calls a “service_a” method, provided by RmiClientM on FW1, RmiClientM invokes the RmiServerM method via RMI, and then calls the “service_a” method provided by CRMMServer on FW2.

We can implement this proxy approach for both CRMMs and OSGi bundles. In both cases, however, each bundle must be activated on the desired cores in proper order. Moreover, in order to dynamically set up the RMI server proxy bundle, the RMI stub class files, as well as other class files that might be loaded in the RMI stub class, must be dynamically stored in the RMI codebase directory, before registration into the RMI name servers.

Through core services, programmers can properly activate each CRMM. In addition, one of CRM core service “extractExecuteModule” command extracts the required class files from the RmiServerM and puts them into the RMI codebase directory automatically, prior to execution. An example of our proposal is shown as follows: We assume that each JAR file is prepared in advance.

1. CRMMClient calls the “executeModule” of CRMMServer on core 2. Then, the CRMMServer is downloaded and executed on core 2, and registers the service_a on FW2.

2. After CRMMClient gets the “executeModule” callback, CRMMClient calls the “extractExecuteModule” of RmiServerM on core 2, with port number argument, managed by CRMMClient. The RmiServerM is executed and waits for the RMI connection request at the specified port, on core 2. Moreover, the RmiServerM gets the service object of service_a, provided by CRMMServer on FW2.

3. After getting the callback, CRMMClient calls the “executeModule” of RmiClientM on core 1, with arguments specifying the IP address and port. Then, the RmiClientM is executed and connects to the RmiServerM towards the specified IP address and port, via RMI. Moreover, the RmiClientM registers service_a to FW1.

4. The CRMMClient can get the service object of service_a from RmiClientM on FW1. The set-up is finished.

5. Finally, CRMMClient can use the service_a provided by the CRMMServer on the remote Core.

Generally speaking, the source codes of RmiServerM and RmiClientM can be written manually. However, we developed a useful tool that automatically creates both of them from the source codes of the service interface. This tool analyzes the methods of the service interface by using Java reflection and creates RmiServerM and RmiClientM class files. This tool is also written in Java.

VI. CONCLUSION

We have proposed a type of communication middleware called CSC, which enables distributed resources to be adaptively coordinated in Internet-based plug-in architecture. It coordinates and arranges the various resources, scattered throughout the end-to-end communication links, by deploying software components at appropriate places. This architecture consists of two fundamental communication control frameworks: CRM for communication resource coordination, and CLM for arranging communication logic. We described the details of CRM, which is implemented on OSGi service platforms. In the description, we mainly focused on two topics; security functions and methods of cooperation between CRMMs. We described and defined the security functions required for CRM and its implementations on OSGi frameworks. Additionally, CRM enables cooperation between CRMMs by sending a string of messages through the CRM core service and using the original OSGi service, either locally or remotely. We succeeded in developing CRM, a network middleware that can provide an environment where downloadable software components can be executed safely and cooperate with each other on distributed OSGi frameworks.

REFERENCES