Reliable Protocol for Robot Communication on Web Services

Masahiko Narita and Makiko Shimamura
FUJITSU LIMITED
masahiko.narita@jp.fujitsu.com

Makoto Oya
Hokkaido University

Abstract

This paper describes the requirements for the reliability of robot communication, which is an important issue for a practical use, especially in a loosely coupled environment, the wireless network and WAN, and explains how to resolve these issues using Web services based protocol "RoboLink Protocol". In order to achieve reliability under these communication conditions, this paper proposes to solve them by combining two solutions: one for the transport layer is adopting the standard reliable messaging technology, and the other for the application layer implements a transaction behavior, with a recovery process for fatal failures. And, it also provides a guideline for developers how to implement these recovery process easily in their application. And more, it reports the implementation strategy and the success of the verification for this solution using our proof-of-concept implementation developed as a plug-in handler of the Web server, and also provides the test cases.

1. Introduction

Robot technologies and industries have been growing rapidly since 1980's, and they contribute greatly to the development of manufacturing. Nowadays, robots start to appear into our daily life space such as the office, our home or in our community. Robot application systems used for daycare, entertainment and community activity are expected soon. However, at this moment, most of the robots have their own interface and communication protocol. Consequently, what robots can do is generally limited to simple tasks that can be done by a single robot. In order to remove this restriction, it is necessary to provide a common interface and protocol connecting robots over the network. The RoboLink Protocol proposed by Narita et al. [1] is one of the attempt for this, and particularly to enable loose communication between robots and computers in an open network environment. The protocol was released by the RoboLink Consortium as a publicly available open standard. Currently, a public draft of version 1.1 is published [2].

Considering the number of technologies and tools supporting Web services – an emerging IT technology -, as well as the development costs and the skill costs associated with it, it is appropriate to use Web services to achieve the control of loose robot communication over the Internet environment [1][3][4]. On the other hand, for a practical use, it is also necessary to ensure the reliability and the security of the communication. Indeed, the use of a wireless network and WAN such as Internet in loosely coupled environments, causes possible failures and quality degradation of the communication.

This paper focuses on the reliability of loose robot communication. It describes the requirements for the reliability of the robot communication in a loosely coupled environment, and explains how to resolve these issues using RoboLink Protocol. Our solution combines the adoption of the reliable messaging standard technology of Web services for the transport layer, with a guideline for application recovery transactions. And more, it reports the implementation strategy and the success of the verification for this solution using our proof-of-concept implementation developed as a plug-in handler of the Web server, and also provides the test cases.

2. The RoboLink Protocol

The RoboLink Protocol is an open protocol for robot communication among entertainment or home-use robots over the network. This protocol is provided by the RoboLink Consortium composed of toy makers, robot makers and IT vendors. Currently, a public draft version 1.1 is published. The main objective of RoboLink Protocol is sparse communication among loosely coupled robots. The RoboLink Protocol has been designed over an infrastructure based on Web services, which is an open technology for communication over the

Proceedings of the 2005 International Conference on Cyberworlds (CW'05)
0-7695-2378-1/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on February 26,2010 at 10:40:40 EST from IEEE Xplore. Restrictions apply.

Internet. Because the Web services technology is mature enough, and already supported by many middleware products and tools, it is also important for robots to be able to leverage these tools as they are.

In order to support various functions for many kinds of robots, the RoboLink Protocol introduces the "Profile" concept. The protocol defines some profiles for basic motions, and functions that are specific to each robot can also be represented as a profile.

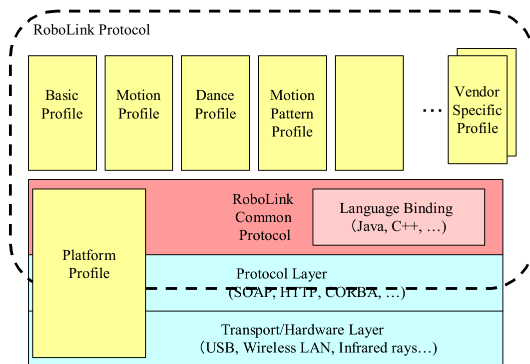

Fig. 1 RoboLink Protocol

By adopting the RoboLink Protocol, several robots can communicate with each other over the network, and can be operated using the same commands. For example, several robots supporting the protocol can be moved one meter forward using a single command. Therefore, robot services, such as home security task and information inquiring, can more easily be supported by using the protocol. A significant initiative for realizing such a network business application with robots is RSi (Robot Services Initiative) [5], which was established in 2004. RSi has done several proof experiments which assume actual use of the services in order to promote the use of such robot services. [6]

3.Reliability Problems And Requirements

In order to control robots in a loosely coupled environment, the reliability of the communication is very important. Because the wireless network and WAN such as Internet are used, some serious trouble may be caused by quality degradation of the communication. According to our findings, confirmed in the previously mentioned demonstration events, four factors stand out regarding failures and quality degradation of the communication. The first one comes from problems caused by the wireless network. In wireless environments used in the broad range, such as wireless LAN and Bluetooth, we observe problems caused by competing access on the same channel, interferences from adjacent access points, interferences caused by microwave ovens or other wireless devices, and decrease in electric field strength with the electric wave interception. The second type of issues is caused by the Internet environment. In an Internet environment such as WAN, problems like the decrease of transmission speed due to rival access, the decrease in processing performance such as relay servers and quality of the line, may occur. The third class of problems is caused by moving robots. A decrease in quality of the wireless line may be caused by a changing distance between robots and the access point and/or by changing the position between robots and the electric wave interception, a consequence of robots moving to different places. The last class of problems comes from robots and/or hosts. Because of some troubles originating in robots or in the systems being used, robots and systems may not be able to communicate with each other. The troubles usually come from the hardware and/or the software.

In order to achieve reliability under the above operating conditions, the following features and functions are required:

(1) Reliability of the transport layer

The reliable transport layer should guarantee message delivery from the sender to the receiver. If the sender and/or the receiver are temporarily down, the reliable transport layer should resend messages automatically. If the same message is received more than once – for example as a consequence of guaranteeing the message delivery -, a reliable transport layer should ignore the duplicated messages. Also, in a sequence of messages, the order of this sequence may have significant meaning. Therefore, a reliable transport layer should guarantee the ordering of the messages.

(2) Detection and recovery of problems caused by moving robots

Because robots are moving, the distance between robots and the access point is likely to vary, and the position between robots and the electric wave interception will also change. This may affect the quality of the wireless line. In such cases, the robot can solve these problems by itself. For example, if a robot notices that the electric wave situation worsens, the robot can move to find a place where the electric wave can be received more easily.

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on February 26,2010 at 10:40:40 EST from IEEE Xplore. Restrictions apply.

(3) Application recovery process

Both the sender and the receiver can independently detect an interruption of the communication. If some trouble occurs during the sending of a message, the transport layer should handle the recovery of the communication. It is not appropriate for the application to take care of resending the message. On the other hand, some troubles may occur, which cannot be handled by the reliability functions of the transport layer alone. For example, a receiver may fail to receive a message in spite of multiple resending attempts from the sender. In this case, one approach is for the sender to give up resending the message. Another is for the sender to send the message again after some time. It is then the role of the application to decide how to resolve this kind of trouble. For the first case, even if a reliable transport layer is used, it is very difficult for application developers to handle this kind of recovery process in their application. We propose a guideline for developers on how to implement such processes in the next chapter.

4. A Solution For Achieving Reliability

The functions of the reliable transport are not peculiar to the robot communication. They are similar to those completed in another field after careful design and significant efforts. They are "Reliable Messaging" functions used to communicate between servers in the IT area. Recently, the reliable messaging function has been standardizing. However, it is difficult to solve the problems in the previous section only with the reliable specification, because the recover processing still remains as programmer's responsibility. Therefore, we solved them by combining two solutions: one for the transport layer is adopting the reliable messaging method, and the other for the application layer implements a transactional behavior and proposed a programmers guideline.

(1) Solution for the Transport Layer

When a failing communication can recover quickly, which is the case for troubles caused by a temporary accident or by a probabilistic factor, a retry mechanism is sufficient. However, if the communication cannot recover after a long time, the trouble cannot be solved by mere retry. In this case, the communication should be cut, and efforts should focus on the removal of the cause of the trouble, as well as on recovery from the interruption point.

An acknowledgement protocol, such as receiving an acknowledgement message (ACK) after sending data, will guarantee that data was sent and received properly. When no ACK is received, the retry for sending data is done at constant intervals, repeatedly until an ACK is received within a specified time. If data reception cannot be confirmed even after many retries, it is assumed that a timeout error will occur. It is necessary to identify data items being sent, so that ACK can refer to the right data item. This identification is also used to detect duplicated data, as when the same data item is received many times due to the resending mechanism. In order to guarantee message ordering, a range or length for the message sequence subject to ordering should be specified, and every data item within the sequence be assigned an order number. When retrying the sending of a message, it is also important to minimize the impact on performance, which could degrade due to frequent resending and/or operating ACKs. If the timeout error occurs during the sending of data, this sending and related operations should be stopped. If the communication is up again, it should resume the sending of data, so that a previously failed unit of data is sent again.

If the other party system is down, the application only knows that a timeout error occurred for the receiver of the message. If the application's system is down, the application may learn about the trouble at that time, and may try to recover. However, usually, the application learns about the trouble after the system is up again. On the other hand, if the session is interrupted by serious troubles, it may not be recovered. In this case, the application learns about the session trouble after a new session is started. Therefore, both the sender and the receiver should keep a copy of a sent/received message, of the message in waiting to be sent, and other message status information. After the system is rebooted, it should handle the sending/receiving of interrupted messages appropriately: either re-send the message, or continue the reception of the message, or cancel the sending/receiving of the message. The retry based function such as of TCP/IP protocol is not enough to make this recovery process.

(2) Solution for the Application layer:

In this section, we address problems caused by serious troubles in the transport layer, and propose a guideline for application programmers so that the development of a recovery process is made easier. To simplify the situation, we assign a server role to a robot and a client role to a controller, and classify the communication between a robot and a controller into the following two typical conversation models (Fig2):

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on February 26,2010 at 10:40:40 EST from IEEE Xplore. Restrictions apply.

a. A client sends a command message to a server (robot). The server executes it and sends a result message to the client. The client waits for this result message before processing further.

b. A client sends a command message to a server (robot), and does not wait for a result message from the server. The client receives a result message from the server when the server completes the execution, or the client regularly polls the status of the server's execution.

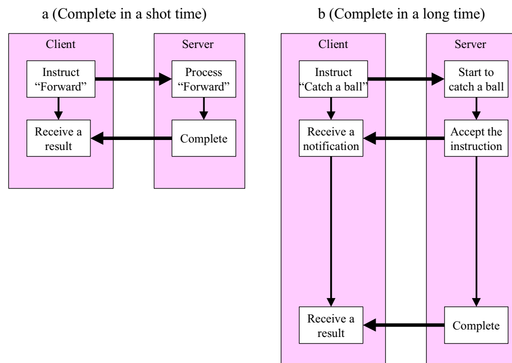

Fig. 2 Conversation Model

Based on the above conditions, if an application needs to recover from a serious error while sending/receiving a message, the application should resolve the following three issues: (i) Ignore all non-processed past messages of the same conversation as the one related to the error message, because they are not current anymore, (ii) Close the executing conversation so that no new message is received for it, (iii) Open a new conversation for the next new messages. For resolving these issues, the application needs to associate messages with conversations. Therefore, we proposed to introduce a unique ID, called ConversationId, which is allocated to every conversation related to an application. The ConversationId should be added to every sending message in both sides and will distinguish a series of operations. Using ConversationId, both a server and a client can recognize messages of a current session, and can detect if they receive messages of a past session. This solution is effective for application to recover, even if a serious error is caused by an interruption of the session, or caused by the problem of client/server.

5. Best Practices

As a proof of concept, we developed a prototype robot application over reliable RoboLink protocol using an implementation of WS-Reliability.

5.1 Advantage of Using a Reliable Messaging Standard

In the area of Web services, OASIS – one of the main standard body for this area -, recently released the "Web Service Reliability" specification as a Standard [7]. There are many advantages in adopting this specification for our purpose. By adopting this specification, robot applications do not need to implement advanced reliable functions such as complex delivery processing, management of resending policies, guaranteed ordering, etc. Developers can concentrate exclusively on functions specific to the robot application. In addition, multiple implementations of the specification are available to choose from, such as the one provided by the joint project of Fujitsu, Hitachi and NEC [8], The University of Hong Kong [9], Easy WS-Reliability [10], etc..

5.2 What is WS-Reliability?

The mainly function of WS-Reliability is to guarantee message delivery without duplicates ("once and only one") and message ordering. If any trouble occurs during the sending of a message, the message will be resent automatically. Fig.3 shows the recovery mechanism when troubles occur during the sending a message.

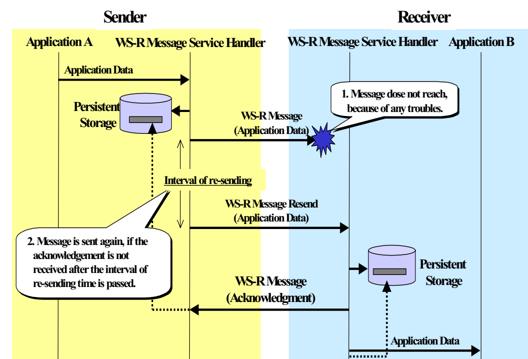

Fig. 3 Recovery Mechanism of WS-Reliability

WS-Reliability has three message reply patterns for returning ACKs or faults: "Response" RM-Reply Pattern, "Callback" RM-Reply Pattern, and "Poll"

0-7695-2378-1/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on February 26,2010 at 10:40:40 EST from IEEE Xplore. Restrictions apply.

RM-Reply Pattern. These patterns can accommodate very diverse communication constraints.

5.3 WS-Reliability and the RoboLink Protocol

In order to use reliable messaging functions, we plugged a WS-Reliability implementation in the lower layer of the RoboLink Service on the SOAP protocol. Our implementation, called OASIS-RM, is developed as a plug-in handler of the Web server, and the way it is operating is conforming with the "Jetty" Java HTTP Server [11], However, the implementation also supports the handler interface, so that other Web servers can be used. We needed to adopt "Response" RM-Reply Pattern as the message reply pattern, because the call from JAX-RPC [12] was the block model.

We can use JAX-RPC as the programming interface. In that case, we had to modify JAX-RPC module to block until sending completion or appearing the serious failure. As a result, the serious operating failures appear as Java exceptions through the JAX-RPC handler and/or as fatal messages from the WS-Reliability implementation, which will be sent to the configured Web server. Below is an explanation on how WS-Reliability processes the fatal errors from the transport layer and switches the transport sessions. WS-Reliability allocates an unique ID, called GroupId, to each session when a new session is opened. If a serious error occurs during the sending of a message, WS-Reliability closes the current session, and opens a new session. During this time, the GroupID used by both sender and receiver is automatically changed.

Messages are identified by their MessageId, and the uniqueness of MessageId is guaranteed by the combination of GroupId and a sequence number, SequenceNum. SequenceNum is assigned by the WS-Reliability implementation, when the message is sent. The receiver must return an ACK to the sender, referring to the MessageId of the sending message. If the sender receives an ACK with an old GroupId, the sender ignores it. When the sender recovers from a sending timeout failure, the sender assigns a new GroupId to a newly-opened session, and generates a MessageId that include the new GroupId for the next message to be sent.

If a message with a new GroupId is received, it means that some serious error occurred on the other side. In this case, the sender aborts the message sending, and updates the current GroupId. If necessary, the sender checks the status of the sending queue, and may delete some data in the queue. Then, the sender recovers the data being sent, because of the possibility of a failure to reach the receiver.

It must be noted that GroupId described here and ConversationId described in the previous chapter may look similar, but play a different role. The former is used by the transport layer for sequences of messages going from a same sending party to a same receiving party, and has reliability semantics. The latter is assigned for each conversation - including messages flowing both ways between parties -, and has application semantics.

5.4 Implementation of application recovery transaction

We verified our recovery strategy using the previous model where a remote controller sends a message to a robot of the server side. For this purpose, the robot-side application was implemented as a service of the Axis Web services stack. The controller-side application was developed as a client application using Axis JAX-RPC client library with the block mode.

Fig.4 shows an implementation design that supports the conversation described in Section 4 (b). The client sends a request, and the server returns the acknowledgment. Then, the server sends the completion notice to the mail box, which is a process distinct from the client. The client polls the mail box directly, and receives the notification from the mail box. In our implementation, the retry function of WS-Reliability was developed as a handler. However, it is out of scope of the AXIS handler to add/delete the SOAP header. Therefore, we modified Apache AXIS.

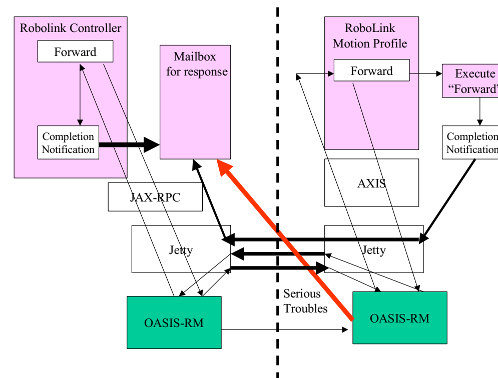

Fig. 4 An sample implementation of the conversation model

5.5 Application Scenario

Proceedings of the 2005 International Conference on Cyberworlds (CW'05)
0-7695-2378-1/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on February 26,2010 at 10:40:40 EST from IEEE Xplore. Restrictions apply.

We developed our applications in the type shown in Section 4 (2) b, and implement the recovery process according to the scenario described in Section 4 (2).

5.5.1 Client Application

The client application has the following sequence to communicate with the server application, shown in Fig.5.

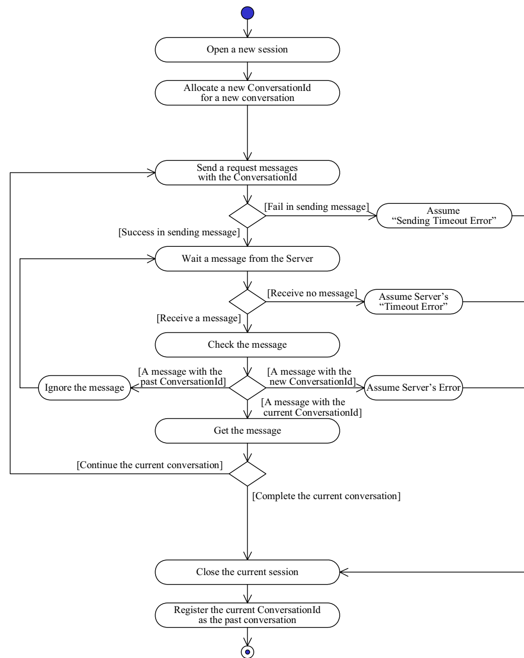

Fig.5: Client application's Action

(i) Allocate a new ConversationId for a new conversation, in order to start a new conversation.
(ii) Send a request messages with the ConversationId to the server application. If the client application cannot send the message, it assumes that the sending timeout occurred. Then, go to (vi).
(iii) Wait a reply message from the server application.
(iv) Check a reply message from the server application.
If a message has the current ConversationId, get the message. If a message has other ConversationId, ignore the message. Then, return to (iii). If the client application receives the "timeout error", it assumes that the server's timeout error occurred. Then, go to (vi).
(v) Determine whether to continue the current conversation.
If the client application wants to continue the current conversation, return to (iii). If the client application wants to finish the current conversation, close the current conversation.
(vi) Register the current ConversationId as the past conversation. Then, returns to (i).

Please note that the client may need to confirm the status of the Server, if the previous session was closed with some error.

5.5.2 Server Application

The sever application has the following sequence to communicate with the client application, shown in Fig.6.

(i) Wait a request message from the client application, in order to start a new conversation.
(ii) Check the message from the client application. If the message has the past ConversationId, get and ignore it. Then, return to (i).
(iii) Wait a request message from the client application for processing request message. If the server application receives the "timeout error", it assumes that the client's timeout error occurred. Then, go to (vi).
(iv) Check the message from the client application. If the message has the current ConversationId, get and process it. If the message has the past ConversationId, get and ignore it. Then, return to (iii). If the message has a new ConversationId, go to (vi) to close the current session.
(v) Determine whether to continue the current conversation.
If the server application wants to continue the current conversation, send the reply message to the client message. Then, return to (iii). However, if the server application cannot send the reply message, it assumes that the sending timeout occurred. And, go to (vi) to close the current conversation. If the server application completes the current conversation, send the result to the client message. And, if the client application cannot send the result, it assumes that the sending timeout occurred.
(vi) Close the current conversation. Then, return to (i).
If the sever application recognizes the current ConversationId, register the current ConversationId as the past session. And, if the message has a new ConversationId, register the new ConversationId as the current session.

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on February 26,2010 at 10:40:40 EST from IEEE Xplore. Restrictions apply.

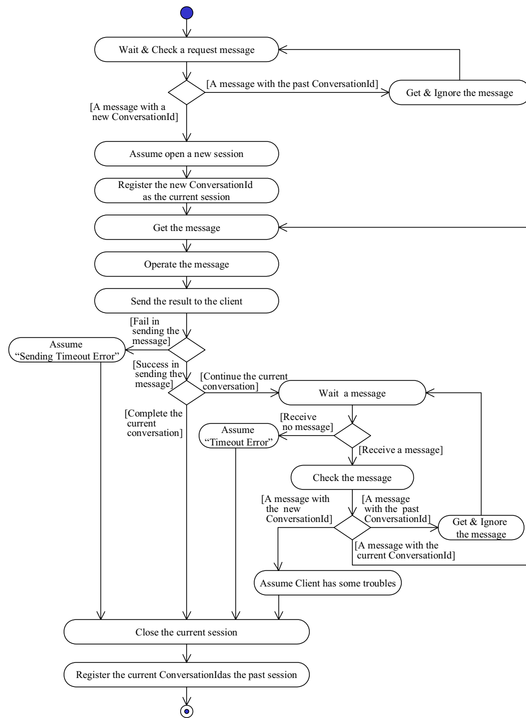

Fig.6. Server Application's Action

8. Verification and Test Cases

We verified the following recovery mechanism using our implementation:

- If communication recovers from a transport trouble before the timeout error occurs, applications can keep sending/receiving messages as if nothing happened.
- After the fetal error occurs, current application sessions are closed, and a new application session is invoked to process a new message. No old message is processed any more.

Our test case is as follows:

No.1: The client sends a request messages to the server, and the server sends the result to the client successfully.

No.2: When the client sends a request message to the server, the server is down. After sending the message again several times, the server is up. Then, the client succeeds, and the server sends the result to the client successfully.

No.3: When the client sends a request message to the server, the server is down. Even if the client sends the message again several times, it fails. The client gives up because of the retry error. ver sends the result to the client successfully.

No.4: When the client sends a request message and the server replies the ACK to the client, some troubles occur. After the client sends the request message again several times, the server succeeds to reply the ACK. Then, the server sends the result the client successfully.

No.5: When the client sends a request message and the server replies the ACK to the client, some troubles occur. Even if the client sends the request message again several times, the server fails to reply the ACK. The client gives up the current message because of the retry error. And, the server execute the request, however, it fails to send the result. Then, the client sends the next request message to the server, and the server sends the result to the client successfully.

No.6: When the client sends a request message to the server, some trouble occurs. After sending the message again several times, the client succeeds. Then, the server sends the result to the client successfully.

No.7: When the client sends a request message to the server, some trouble occurs. Even if the client sends the message again several times, it fails. The client gives up the current message because of the retry error. Then, the client sends the next request message to the server, and the server sends the result to the client successfully.

No.8: The client sends a request message to the server, and the server sends the result successfully. The client sends the next request message to the server, the server is down. While the client sends the message again several times, the server is up, and a new session is opened. Then, the clients sends the message again, and the server sends the result successfully.

No.9: The client sends a request message to the server, and the server sends the result successfully. After that, the client sends the request message to the server, the server is down. Even if the client sends the message again several times, it fails. The client gives up the current message because of the retry error. Then, the client sends the next request message to the server, and the server sends the result to the client successfully.

No.10: After the client sends a request message to the server, some troubles occur when the server sends the result to the client. After sending the result again several times, the server succeeds.

No.11: After the client sends a request message to the server, some troubles occur when the server sends the result to the client. Even if the server sends the result again several times, it fails. The server gives up to send the result because of the retry error. And, the client also gives up the

Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on February 26,2010 at 10:40:40 EST from IEEE Xplore. Restrictions apply.

current message because of the timeout error. Then, the client sends the next request message to the server, and the server sends the result to the client successfully.

No.12: When the server sends the result to the client, the client is down. Even if the server sends the result again several times, it fails. The server gives up to send the result because of the retry error. Then, the client is up. The client opens a new session and sends the next request message to the server . The server sends the result successfully.

No.13: The client sends a request message to the server. It takes longer time for the server to execute the request message, the client's timeout error occurs. Then, the client sends the next request message to the server. The server execute the second message, after the execution of the first message is completed. The server sends the result of the second message to the client successfully.

No.14: The client sends a request message to the server. It takes longer time for the server to execute the request message, the client's timeout error occurs. Then, the client sends the next request message to the server. However, the server doesn't complete the execution of the first message. Since the client cannot receive the result, the timeout error occurs again. Then, the client sends the next request message to the server again, and the server sends the result to the client successfully.

9. Conclusions

We described the importance of the reliability of the robot communication in a loosely coupled environment using Web services. And, we proposed the solution that combines the adoption of the standardized reliable messaging technology "WS-Reliability" and the transaction behavior for the application layer implementation. And more, we also proposed the guideline for application recovery transaction. Our implementation worked according to the recovery mechanism that we expected, and the solution was verified through analyzing WS-Reliability and testing our proof-of-concept implementation.

The reliability of robot communication was identified as the very important issue for a practical use in a loosely coupled environment. And, we proposed the solution to achieve reliable communication by combining standardized Web services technology and application recovery transactions. Web services are mature technology, supported by many middleware and tools. It is important for future robots to be able to use these as they are. In order to verify this solution, we developed the sample implementation.

The loosely coupled protocol described here is suitable for applications involving autonomous robots. Moreover, when reliable communication is realized as described in this paper, it is easy to provide an interface to the upper-layer application of the robot that is based on Web services. We expect that various robot services based on Web services will be provided for future business and industrial application of robots.

10. Acknowledgment

Authors of this article would like to thank Akiyoshi Katsumata of Fujitsu and Jacques Durand of Fujitsu Software Corporation for his contribution to produce this document.

11. References

[1] M. Narita, K. Naruse and M. Oya, "RoboLink: A Robot Collaboration Protocol based on Web Services", *ICAM'04*, JSME/No.05-204, ISSN1348-8961, pp.442-448, 2004.
[2] M. Narita and etc, "RoboLink Protocol Specification Public Draft v1.1", http://www.osl.fujitsu.com/osl/contents/RoboLink/RoboLinkProtocol11.pdf (in Japanese)
[3] M. Oya, K. Naruse, M. Narita, T. Okuno, M. Kinoshita, and Y. Kakazu, "Loose Robot Communication over the Internet", *Journal of Robotics and Mechatronics*, Vol.16, No.6, pp. 626-634, 2004
[4] M. Oya, K. Naruse, M. Narita, T. Okuno, K. Moni, M. Kinoshita, and Y. Kakazu, "Loose Robot Collaboration in the Public Internet Environment", *ICAM'04*, JSME/No.348-8961, ISSN1348-8961, pp.454-461, 2004.
[5] Robot Service Initiative (RSi), http://www.robotservices.org/
[6] M. Narita and M. Shimamura, "A Report on RSi (Robot Services Initiative) Activities, *ARSO '05*, 0-7803-8948-4/05/$20.00, 2005
[7] OASIS Web Services Reliability (WS-Reliability V1.1), http://docs.oasis-open.org/wsrm/2004/06/WS-Reliability-CD1.086.pdf
[8] RM4GS (Reliable Messaging for Grid Services), Information-technology Promotion Agency, Japan, http://businessgrid.ipa.go.jp/rm4gs/index-en.html
[9] Web Services Reliability Messaging Server, Center for E-Commerce Infrastructure Development (CECID), http://www.cs.hku.hk/%7Efyp001/index.html
[10] Project: Easy WS-Reliability, http://sourceforge.net/projects/easywsrm/
[11] Jetty, http://jetty.mortbay.org/jetty/index.html
[12] Java API for XML-based RPC (JAX-RPC1.1) http://java.sun.com/xml/downloads/jaxrpc.html

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on February 26,2010 at 10:40:40 EST from IEEE Xplore. Restrictions apply.