LETTER

Towards Applying Dynamic Software Updating for DDS-Based Applications*

Dong Kwan KIM^{†a)}, Member, Won-Tae KIM^{††}, and Seung-Min PARK^{††}, Nonmembers

SUMMARY In this letter, we apply dynamic software updating to long-lived applications on the DDS middleware while minimizing service interruption and satisfying Quality of Service (QoS) requirements. We dynamically updated applications which run on a commercial DDS implementation to demonstrate the applicability of our approach to dynamic updating. The results show that our update system does not impose an undue performance overhead—all patches could be injected in less than 350 ms and the maximum CPU usage is less than 17%. In addition, the overhead on application throughput due to dynamic updates ranged from 0 to at most 8% and the deadline QoS of the application was satisfied while updating.

key words: dynamic software update, data distribution service, software evolution, quality of service

1. Introduction

The OMG Data Distribution Service (DDS) specification [1] defines a publish-subscribe middleware, which enables distributed nodes to share data, events, and commands coming from the nodes at short intervals while satisfying user's QoS requirements. With the publish-subscribe model, a publishing node does not need to identify where data objects are sent and a subscribing node is not concerned with the data source, which provides the values of the data objects. The publishing nodes list up the data objects they will publish, and the subscribing nodes identify which data objects they are interested in. The DDS middleware is intended to support distributed systems like net-centric warfare systems that are long-lived, large-scale, real-time, and embedded [2]. Since such systems require high availability and the system's redeployment is nontrivial, it is preferable not to reboot them frequently to change part of the whole system (e.g., bug fixes, security patches, and addition of functionality).

Dynamic Software Update (DSU) [3]–[6] addresses these concerns by updating the code of a computer program while it runs without significant performance overhead. This letter applies the DSU technique to DDS-based applications—a computing domain characterized by long-

Fig. 1 DDS-based applications with dynamic update mechanism.

running operations and high reliability. The proposed update system provides several features that would be practical and useful to update DDS-based applications. Our update system does not need to preprocess target applications to make them updateable using code instrumentation or annotation at source code or binary level. In addition, the system generates dynamic patches by comparing the old and new versions of object and binary files without analyzing source code. Furthermore, our DSU system supports rollbacks of committed patches that make it possible to go back to before updating.

Figure 1 depicts a DDS-based application with dynamic update systems. Publishers and Subscribers share the specified information (e.g., update messages) through the DDS facilities such as flow control, automatic discovery, and QoS policies [7]. Before data distribution, some Publishers and Subscribers register their interesting data, called *Topic*, which defines data structure to be sent and received over the network. Every Publisher can generate dynamic patches that contain the differences between the old and new versions of a program. The *Topic, Update Message* shown in Fig. 1 includes patch information such as patch names and patch locations. The Subscribers that are of interest in the patch will receive the *Update Message Topic*. Upon receiving the patch, the Subscriber initiates the Update Mechanism to make changes to the running itself. Finally, the Update Mechanism replaces the old version with the new one.

2. Proposed Dynamic Updating System

We developed a practical dynamic update system that can change a running program loaded in memory to perform updates without program restart.

Manuscript received April 18, 2011.

Manuscript revised October 31, 2011.

[†]The author is with the Department of Multimedia Engineering, Catholic University of Pusan, Pusan, Korea.

^{††}The authors are with the CPS Research Team, Embedded SW Research Dept., ETRI, Daejeon, Korea.

*This work was supported by the IT R&D Program of MKE/KEIT [10035708, "The Development of CPS (Cyber-Physical Systems) Core Technologies for High Confidential Automatic Control Software"].

a) E-mail: dongkwan@gmail.com

DOI: 10.1587/transinf.E95.D.1151

Fig. 2 Architecture of the proposed dynamic update system.

2.1 Generating Patches

Figure 2 shows our dynamic update system that consists of a patch generator and a dynamic update executor. The patch generator identifies the differences by comparing the old with new versions of a target program. To minimize the performance overhead due to dynamic updates, the patch generator constructs the size of a patch file as small as possible. The major elements of the patch generator are an object/binary file analyzer, a symbol relocation checker, and a patch file creator.

The object/binary file analyzer takes object and binary files of the two versions to extract the differences between them. For generating dynamic patches, the object/binary file analyzer obtains the necessary update information such as binary file names, execution symbols, and object file names. The update system cares for the modification of existing variables and functions and the addition of new variables and functions. When variables and functions have been changed, their symbol names remain the same, but their symbol information could be altered in the new version. For the modification updates, the object/binary file analyzer retrieves such symbols that have the identical name and different symbol information and then stores the symbol information and code. In the case of the addition updates, the symbols for the added variables and functions are extracted from the new version and their symbol information and code are stored.

Some updated symbols need to be relocated due to the difference between memory addresses allocated in the binary file and physical memory ones. The symbol relocation checker extracts information about the relocation symbols from the object files and stores the names and types of these relocation symbols. Special care should be taken for read-only data in the binary file because they have no assigned symbols. The symbol relocation checker cannot get relocation information regarding the read-only data from the object files. To address this matter, our dynamic update system extracts the read-only data in a function and then attaches them to the next of the function code.

To generate a patch file, the patch file creator uses update information and symbol relocation information. The

patch file creator describes the header information of the patch file, initializes all sections that will be added to the patch file, creates the section information, produces the delta image that contains the changes of the target code, and finally stores the delta image and the section information to the patch file.

2.2 Applying Patches

The dynamic update executor shown in Fig. 2 injects a patch file to the program on the fly. It consists of a patch analyzer, a patch loader, a symbol relocater, and a branch code injector. To extract update information from a patch, the patch analyzer analyzes the patch header to assort all sections in the patch according to functions. The patch loader will load the classified patch into reserved memory area. If any symbol needs to be relocated, the patch analyzer stores the information for symbol relocation. The symbol relocater will use that information to relocate the related symbols.

The patch loader dynamically allocates memory storage for use in the patch and loads that patch code into the memory area. We somewhat modified the standard Linux system call, *mmap(3)* to enable a process for our update system to allocate memory space for another at runtime. The patch loader obtains unused heap memory of the updated program and allocates that memory area to the patch code using the modified *mmap(3)*. The patch code is loaded by the *ptrace()* system call. For update rollback, the patch loader stores the information about the allocated memory area.

After the changed code is loaded into unused memory space of the running program, some symbols should be relocated so that the loaded patch code can be executed correctly. To locate the memory address of the relocatable symbol, the symbol relocater computes the relative addresses and finds the absolute addresses from a symbol table. To allow the update to take effect, all function calls to an old function should be delegated to a new function. The branch code injector inserts a branch instruction like *jump* into the entry point of the old function to delegate the function calls.

3. Changing DDS Applications at Runtime

We show the applicability of our approach to dynamic updating via benchmarks and case studies of updating DDS-based programs. We conducted our experiments on Intel(R) Core(TM) i7 PCs with 2 GB of RAM, connected by a 100 Mbps Fast Ethernet network. The systems was configured with OpenSplice DDS version 5.3 [8] on Ubuntu 10.04, kernel version 2.6.32.

3.1 Performance Evaluation

To show the impact of online update, we measured the execution time for applying updates to a running system. We dynamically changed an updated application from its old version to newer one by varying the patch size. Figure 3

Fig. 3 Performance overhead on applying patches.

Fig. 4 Measuring the amount of CPU usage by varying the patch size.

illustrates the elapsed time against the size of the associated patch. The update time includes the whole process of the dynamic update—loading the patch into memory, injecting a branch instruction into the entry point of an old function, and making the patch to take effect. All updates shown in Fig. 3 are completed within less than 350 milliseconds. We can see that the time to apply an update seems relatively modest and our update system does not cause significant performance degradation. Though not exactly, the update time increases linearly with the size of the binary patch, which is roughly related to the number of changes to variables and functions.

Figure 4 presents the amount of CPU usage consumed by the live updates. We measured the amount of CPU time for a normal case and four updating cases by varying the patch size: 92 KB, 253 KB, 380 KB, and 470 KB. The CPU usages are measured ten times at the every one second over ten seconds. The results show that the maximum CPU usage of each case is 3.5%, 5%, 11%, 15.15%, and 16.16%, respectively. These measurements indicate that the CPU usage increases roughly proportionally to the size of the patch file.

3.2 Impact on DDS QoS Policies

DDS messages can be distributed with a specific QoS policy which can be set on Topics, Data Readers, and Data Writers. Since the QoS policy defines the expected behavior when sending and receiving messages, it is considered as a mutual agreement between a Publisher and a Subscriber. Therefore, dynamic updates for a DDS application should not violate the predefined QoS policies.

Fig. 5 Impact of dynamic update on *deadline* QoS for Data Writer on Publisher.

Fig. 6 Impact of dynamic update on *deadline* QoS for Data Reader on Subscriber.

To demonstrate the impact of dynamic updates on running DDS applications, we measured the delayed time on sending messages to Data Readers when updating a Publisher program with accompanying Data Writers. Figure 5 depicts the impact on dynamic updates on the *deadline* QoS for Data Writers during the process of updating. The *deadline* QoS policy defines the period within which a new message is sent by Data Writers or to be received by Data Readers. In Fig. 5, we assumed that data collected from a sensor are delivered with a period of 25 milliseconds to a DDS middleware. To deliver the messages without a loss, a Data Writer is expected to send them within the deadline, 25 milliseconds. Figure 5 shows ten messages (from *msg1* through *msg10*) from the sensor at every 25 milliseconds and the elapsed time to write these messages to Data Readers. The horizontal dashed line on the top of the figure indicates the maximum time allowed to elapse before a new message is sent. We dynamically updated a Publisher application with a Data Writer by applying a patch file of size 328 KB. The update time is 113 milliseconds in which it covered four messages from *msg5* to *msg8*. As seen in Fig. 5, the times to write two messages, *msg7* and *msg8* are greater than ones to write the other messages. Such experimental results indicate that *msg7* and *msg8* were affected by the update. However, the increase in the elapsed time to write the messages is still modest and acceptable because the elapsed time is less than the predefined deadline. In addition, we measured the impact on dynamic updates on the *deadline* QoS for Data Readers on a Subscriber program. We assumed that the deadline on a Subscriber is set to 60 milliseconds. Figure 6 shows four messages (from *msg5* through *msg8*)

Fig. 7 Service Interruption during updating.

among ten messages are received during updating and all messages meet the deadline.

If dynamic updates interrupt a running DDS application, the throughput of the application would decrease as it could miss some messages during updating. In this context, throughput is the rate of successful message delivery between a Publisher and a Subscriber on the DDS middleware. Figure 7 depicts the rate of successfully delivered messages for one second. The Publisher sent a total of 185 messages for one second and the Subscriber was dynamically updated for 97 milliseconds. Since the throughput decreases by a small amount (about 8%) during applying the dynamic patches, our update system imposes negligible service interruption while updating.

3.3 Discussion

An updateable application should provide both the normal and updating operations. While the normal operation focuses on realizing a set of functional requirements, the updating operation enhances runtime adaptability of the application. Although the updating operation allows the application to become reliable at runtime, there is a trade-off between the normal and updating operations. The updating operation enables the programmer to modify DDS-based applications on the fly, thus saving the programmer's time and computing resources. DDS-based applications will be able to exchange messages seamlessly between distributed nodes with minimal service interruption. However, this additional operation could lead to performance degradation on the normal execution of the program. We conducted the experiment to demonstrate the effect of the updating operation on the normal one in terms of performance overhead. Figures 5 and 6 show that the updating operation increases the time required for message delivery. The updating operation delays the delivery of messages *msg7* and *msg8* in Fig. 5 and *msg7* in Fig. 6. When it comes to CPU consumption, Figure 4 presents all the updating operations show higher CPU usage than that of the normal operation. In addition, Figure 7 shows the throughput of message delivery before, during, and after dynamic updating. During the update (from 133 to 150 msgs), the throughput decreases by a small amount because the updating execution interrupts the normal one. The experimental results mentioned above show that our updat-

ing system is practical and relatively efficient in spite of its performance degradation.

To satisfy the *deadline* QoS, we need to consider some design decisions in developing updating systems. The CPU overhead of the proposed update system stems from loading patch contents into memory, doing necessary preparation work, and injecting the branch instruction. Such updating tasks involve system calls such as *ptrace()* and *mmap()*. In spite of the advantages of using system calls, they could interrupt and slow down the normal execution of the program. Therefore, to avoid unacceptable delay due to dynamic updates, we need to minimize the use of system calls that could increase the elapsed time to write or read DDS messages. We also need to create patch files as small as possible to improve the QoS of the DDS middleware. Figures 3 and 4 illustrate that as the patch size becomes larger, it takes more time to load the patch into memory. In addition, when the updating operation exceeds the acceptable execution time, the user can reject the update request to satisfy the *deadline* QoS by triggering an event for the update rollback.

4. Conclusions

We presented an approach that updates distributed DDS-based applications at runtime. The experimental results show our update system does not cause significant performance degradation and does not consume a lot of CPU resources. In addition, the update system satisfied QoS requirements for data delivery while updating. Therefore, our approach to dynamic updating is viable and applicable for updating DDS-centric applications. As the support of seamless service and reliability has become an indispensable part of DDS-based systems, we believe dynamic software updating can benefit software developers in maintaining these systems.

References

- [1] OMG, "Data distribution service for real-time systems," <http://www.omg.org/cgi-bin/doc?formal/07-01-01>, 2007.
- [2] N. Wang, D.C. Schmidt, H. Hag, and A. Corsaro, "Toward an adaptive data distribution service for dynamic large-scale network-centric operation and warfare (NCOW) systems," Proc. IEEE Military Communications Conference, pp.1-7, Nov. 2008.
- [3] M. Hicks and S.M. Nettles, "Dynamic software updating," ACM Trans. Programming Languages and Systems, vol.27, pp.1049-1096, Nov. 2005.
- [4] K. Min, K. An, I. Jang, and S. Jin, "A system framework for map air update navigation service," ETRI Journal, pp.476-486, Aug. 2011.
- [5] K. Makris and R.A. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction," USENIX 2009, June 2009.
- [6] D.K. Kim, E. Tilevich, and C.J. Ribbens, "Dynamic software updates for parallel high-performance applications," Concurrency and Computation Practice & Experience, vol.23, pp.415-434, March 2011.
- [7] D.C. Schmidt, A. Corsaro, and H. Hag, "Addressing the challenges of tactical information management in net-centric systems with DDS," CrossTalk: The Journal of Defense Software Engineering, vol.21, pp.24-29, March 2008.
- [8] OpenSplice DDS, <http://www.opensplice.org/>