Half-Push/Half-Polling

Youngsu Son
Home Solution Group
Samsung Electronics
arload.son@samsung.com

Jinho Jang
Dept. of Computer Science
Hanyang University
jinhoyo@nate.com

Jemin Jeon
Dept. of Computer Science
Hanyang University
luvjjm@gmail.com

Sangwon Ko
Dept. of Computer Science
Hanyang University
funkcode@gmail.com

Hyukjoon Lee
Dept. of Computer Science
Hanyang University
xenoplo@gmail.com

Jungsun Kim
Dept. of Computer Science
Hanyang University
jskim@cse.hanyang.ac.kr

ABSTRACT

To create constantly evolving software, the upgrading is an essential factor. There are two ways to upgrade, pushing and polling. Polling has the advantage of keeping the latest versions of all the clients, but can cause heavy server load by simultaneously connections to many clients and unnecessary network traffics. On the other hand, push causes much less because push can upgrade the specific clients, but there is cumbersome monitoring to keep stopped clients on latest version. The Half-Push/Half-Polling pattern mixes these two different ways, keeping their advantages, eliminating their disadvantages.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features – *patterns.*

D.2.11 [**Software Engineering**]: Software Architectures – *patterns.*

General Terms

Algorithms, Design, Reliability.

Keywords

Upgrade Ticket, Push/Pull Updater

1. INTRODUCTION

Software must reflect the changes of the real world. If not, software will decays over time. No matter how well-made, software must always be revised based on client's requirements changes and the needs for new services. Due to it, many applications currently support upgrade services in order to improve the customer satisfaction. Through this upgrade service, the clients can use the latest services without installing a new program on every time.

One of the major issues to consider on upgrade service is data transmission method. In the majority of cases, server resources are limited. For this reason, an efficient data transmission method is needed.

In client/server model, polling and pushing are generally used as data transmission methods. However, the polling method can cause server overhead when many clients request the upgrades simultaneously. And the push method has problems such as failure to complete the upgrade when the client is offline or when errors occur while upgrading. So the clients that did not upgrade must be managed to be upgraded in later time.

A more efficient data transmission method, therefore, is needed to make up for these weak points. In this paper, we present the Half-push/Half-polling pattern which mitigates the weak points of polling and push methods. The pattern reduces the server overheads. Using distributed update time and scheduling, it also applies suitable upgrades from considering each client's features.

2. BACKGROUND

In Client-Server Model, Push or Polling are general ways to update clients. Push refers to actively updating clients by the server. The server requests access to the clients and pushes data to them. The advantage of this approach is that you can fully utilize the server's resources within the limit of network bandwidth. However, this method assumes that the client will always be alive. And it also requires the server to keep information about clients.

The figure1 shows how Push method works. When some events(ex; upgrade) occur and the server needs to connect to the clients, the server can control the workload autonomously in consideration of its capacity and the bandwidth. Numbers in the picture represent a sequence of operations.

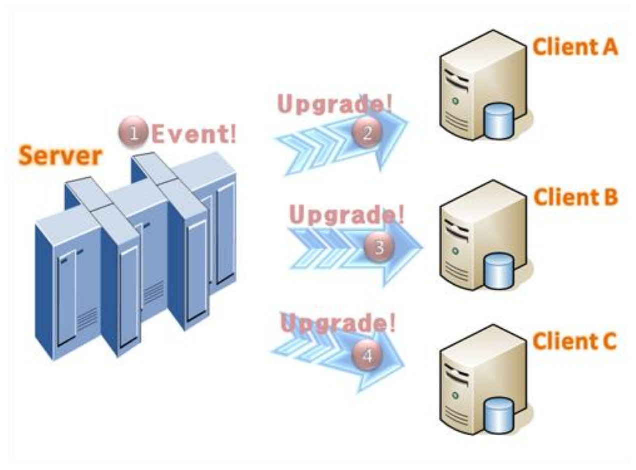

Figure 1. Push Upgrade Method

On the other hand, when using polling method, the clients have the initiative. The clients request data transmission from the server. Because the server is more stable and using this method guarantees the server is alive, the chance of the problem is very low.

However, in Polling method, as opposed to Push, the clients check repeatedly whether any events have occurred or not. And all of clients would attempt to access to the server at the same time in the worst case because there is no fixed order. The figure below shows such case.

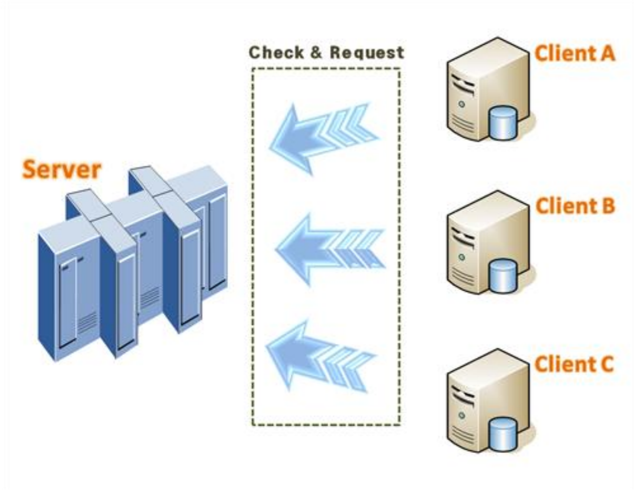

Figure 2. Poll Upgrade Method

3. EXAMPLE

For example, let's assume that we are developing an office automation system for buildings located closely together in a downtown. There are various types of devices in the system and they are connected to a wired or wireless network. In addition, requirement is to keep the software in each device up-to-date. The server will provide the latest software via client-server model. In Polling method, each client requests data from the server without considering any other clients. So, it would cause server overload.

On the other hand, In Push method, a client that is turned off or malfunctions at the time of the Push wouldn't be updated.

Let's take a specific example with the figure below. The server has to update various types(green, yellow, red) of devices that are placed in different location. Some office would have all types of devices but some would not. In this situation, it is possible for the server to manage devices in a way that groups them by device type or location.

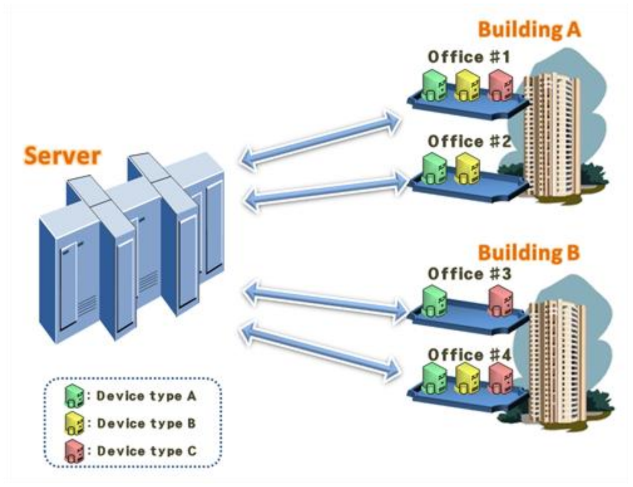

Figure 3. Office Automation

4. CONTEXT

The Upgrade system needs a lot of servers if all of clients must be upgraded as soon as possible, such as anti-virus programs.

However, in the case where there is no time limitation, like a Windows Update, the server can update clients easily within the limits of its resources regardless of the state of clients.

5. PROBLEM

Imagine we are building a data transmission system to send the newest version continuously to the clients. In the client/server model, polling and push methods are generally the used data transmission methods for limited server resource. However, the polling method can cause server and client overhead when many clients request the upgrades simultaneously and check server version periodically.

If we adopt push method on the system, we can get the benefit of reducing the overhead used for the scheduling. But push method has problems when the client cannot complete the upgrade, for example when the client is off-line or it encounters an error during upgrading. We, therefore, need an efficient and reliable data transmission method to maintain the condition of clients using scheduling methods.

6. FORECES

The following items should be regarded as forces:

- Consider that the server has limited throughput to make upgrade possible.
- It should be possible to manage various clients by group.

- Clients who fail during the upgrade can upgrade in the future.
- We need to balance efficiency with reliability.

7. SOLUTION

The half-push/half-polling pattern overcomes the disadvantages of upgrading based on either push and polling method.

Pusher, the upgrade server, uses schedules to distribute the upgrade time to Pollers(clients) which are on the upgrade list. At distributed time, the client requests upgrade from the pusher which then executes the upgrade service. Therefore we can avoid the Non-Stop Talker which polling method invokes. Normally to overcome the disadvantage of push methods, the clients who are off-line have to be managed separately. But in our pattern, at the time when those clients are booted, the clients request their upgrade time. Because of that, we don't't' need to manage the clients separately.

7.1 Structure

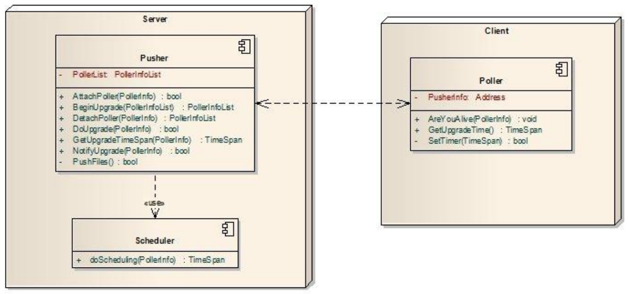
Server
Pusher
- PollerList: PollerInfoList
+ AttachPoller(PollerInfo) : bool
+ BeginUpgrade(PollerInfoList) : PollerInfoList
+ DetachPoller(PollerInfo) : PollerInfoList
+ DoUpgrade(PollerInfo) : bool
+ GetUpgradeTimeSpan(PollerInfo) : TimeSpan
+ NotifyUpgrade(PollerInfo) : bool
- PushFiles() : bool
«use»
Scheduler
+ doScheduling(PollerInfo) : TimeSpan
Client
Poller
- PusherInfo: Address
+ AreYouAlive(PollerInfo) : void
+ GetUpgradeTime() : TimeSpan
- SetTimer(TimeSpan) : bool

Figure 4. Half-Push/Half-Polling Structure

Pusher component is the advanced component that improves existing push functions. This component keeps a list of Pollers that demanded an upgrade and allocates upgrade time.

- AttachPoller – This function registers new Pollers(clients). The goal of this function is to add Pollers to PollerList, the scheduling target list.
- DetachPoller – This function is for deleting Poller from PollerList.
- BeginUpgrade – Entrypoint to start the upgrade. It is the exposed interface to the external source which requested the upgrade.
- NotifyUpgrade – Pusher notifies the upgrade process to selected Pollers.
- GetUpgradeTimeSpan – Poller calls this function to assign the upgrade time from pusher after receiving the upgrade request through NotifyUpgrade from pusher.
- DoUpgrade – To call PushFiles() for upgrade when the assigned upgrade time of Poller is 0.
- PushFiles – To forward the actual upgrade list of files to Poller.

Poller is a component which needs the upgrade regularly. It has to have the right Pusher address that Poller can access all the time.

- GetUpgradeTime – Receive the upgrade time from Pusher.
- SetTimer – Set the time assigned from Pusher. However, it does not run the upgrade directly at the assigned time. Instead of that, the Poller calls Pusher's GetUpgradeTimeSpan() function to provide an upgrade window in case the server's throughput would reach the limit.

The scheduler implements a scheduling strategy for distributing the upgrade time. The scheduler component can use different scheduling strategies[10][11] for different clients(Pollers).

7.2 Dynamics

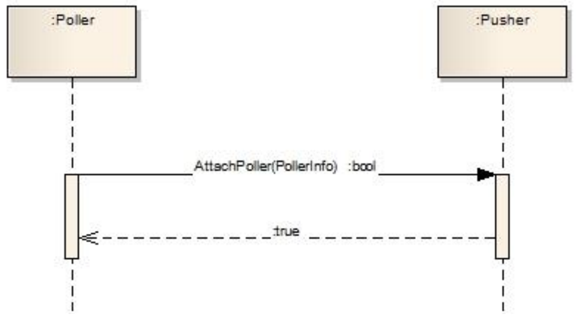

Figure 5. Registration

As in Publisher-Subscriber[7], the Poller(client) can register or cancel the execution of the upgrade service by passing its own reference to the Pusher(server).

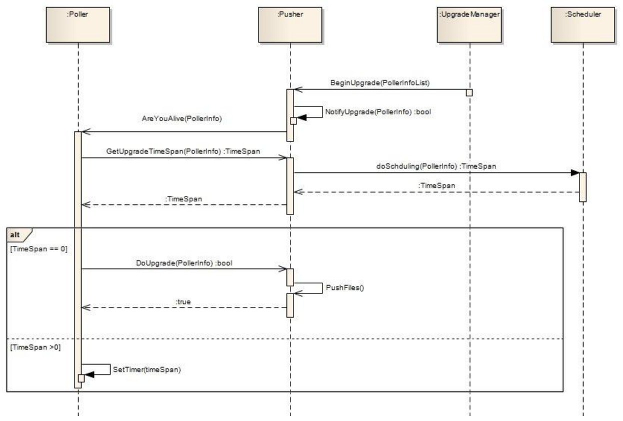
:Poller
:Pusher
:Upgrade Manager
:Scheduler
BeginUpgrade(PollerInfoList)
NotifyUpgrade(PollerInfo) :bool
AreYouAlive(PollerInfo)
GetUpgradeTimeSpan(PollerInfo) :TimeSpan
doSchduling(PollerInfo) :TimeSpan
:TimeSpan
:TimeSpan
alt
[TimeSpan == 0]
DoUpgrade(PollerInfo) :bool
PushFiles()
:true
[TimeSpan >0]
SetTimer(timeSpan)

Figure 6. Upgrade Sequence Diagram

The BeginUpgrade() function is executed by the server application to Pusher.

Then, the Pusher sends the upgrade information(version etc.) through AreYouAlive() function to Pollers. The Poller calls the GetUpgradeTime() function when it needs the upgrade after comparing its version with the pusher's version. After that, the scheduler is called for allocating the upgrade time interval internally. If the upgrade time, TimeSpan, is set as specific hour such as 3:04 Pm, a complex time synchronize mechanism[2][9] is needed between Pusher and Poller or Pollers. To avoid that, we set the TimeSpan as time interval such as 300ms, 400ms.

When the assigned upgrade time becomes 0, Poller calls the DoUpgrade() function.

The Pusher creates a list of files and transmits it to the Poller by calling PushFiles(). The Poller doesn't take files itself. Instead, the Pusher sends them to the Poller in the form of notification. The reason for this is to obtain flexibility that enables providing different file information to the Pollers depending on version – even if they have same type.

But if the assigned upgrade time is more than 0, the Poller waits until it reaches 0. When it reaches 0, it does not run the upgrade directly. Instead, Poller calls pusher's GetUpgradeTimeSpan() function in case of server's throughput would reach the limit. In such case of server's throughput reaching the limit, the Poller would get a new upgrade time for server availability.

8. IMPLEMENTATION

Step 1 : Collect the characteristics of upgrade targets(Pollers).

We could consider the clients (Pollers) that always connect to the network as the upgrade target. However, we should adopt the polling method for clients who are not often connected to the network, such as P2P.

Step 2 : Choose the scheduling algorithm considering Quality of Service(QoS).

The following QoS[6]should be taken into the consideration when performing–the upgrade time.

- Flexibility; any part of system can be upgraded.
- Robustness; the risk of error and crash should be minimized.
- Ease of use; upgrade process should be concise.
- Low overhead; it should minimize the impact on system performance.
- Cost; it should minimize the cost of upgrading
- Independence; modules which are not related to upgrade should not be considered.
- Reliability; Robustness is related to risk during upgrade, Reliability is related at the end of the upgrade. In other words, it should be able to trust that upgrade is done correctly.
- Integrity; it should maintain one status for upgrade, complete or nothing. Upgrading only part of files should not be allowed. This means rollback function should be supported when an error occurs during the upgrade.
- Continuity; the upgrade should be run without interruption.

It is important to understand QoS for the upgrade. The right scheduling strategy is determined according to the system. If the system deals with a variety of upgrade versions or deadlines, the scheduling becomes important issue in a system like real-time system.

Step 3 : Decide the message exchange format.

If a standard message format is used, such as XML for interoperability, a conversion process is required like Marshling/Unmarshaling. This is not suitable for a system that requires quick responses due to the limited resources. In that case, despite system dependency, we should require a message transfer format based on protocols using Binary Method Table[7], such as COM+, OLE, for performance guarantee.

Step 4 : Information from Poller to Pusher should allow extension.

The information sent to Pusher can be changed according to the scheduling algorithm or Poller's feature. Various Pollers are added in the system and thus, the exchange information is designed for extension. For that reason, it should consider using – the Composite Message[1] or the Parameter Object pattern[3].

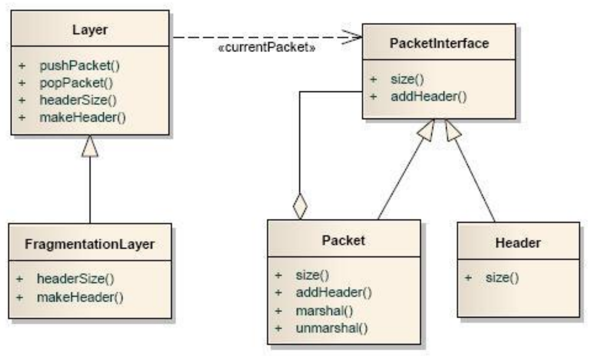

Figure 7. Composite Message Pattern

In the Composite Message pattern, the exchange message can be added/removed through Pipes or Filters. And then, it exchanges or extends protocol easily. Due to these flexible structures which slow down the process, it is not suitable for embedded systems which have limited resources and requires quick response.

Step 5 : Consider type, group, or kind of dependencies between the Pollers.

In most cases, the system has various Pollers(For example, the company has many different kinds of mp3 devices or the company manages various version of electronic products). On top of it, there are dependencies among Pollers for certain services. In this case, to solve the problem, we can make various groups using Event Channel[10] between Pusher and Poller as in figure 8.

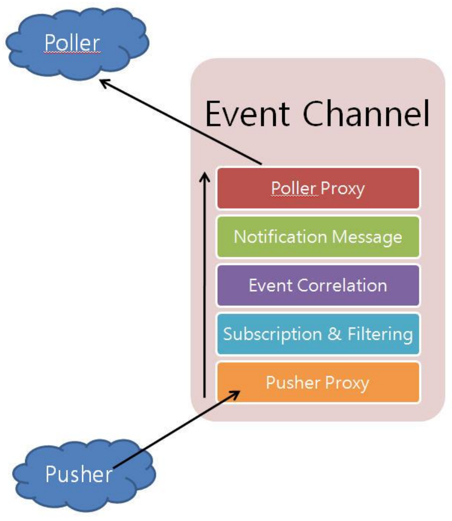

Figure 8. Instance of Event Channel

```
class EventChannel
{
   ……….
   // Managed poller list
   private List ConsumerList;
   private List FilterList;

   public bool AttachConsumer();
   public bool DetachConsumer();

   //assign the filter for grouping of various pollers(upgrade
   target)
   public bool AttachFilter();
   public bool DetachFilter();

   //Start the EventChannel service.
   public bool Run();

   //send the poller's information ( i.e. device own ID, state
   information, App version, Framework version information etc.)
   public void NotifyUpgrade();
};
```

The picture and source above are the examples of Event Channel structure.

The direct dependency between Pusher and Poller is removed because various Pollers are managed with Event Channel as above. In other words, this is a more flexible structure–to allow changes when adding/removing new Pollers or when targeting the upgrade groups. In addition, using the filters, the upgrade targets(Pollers) can be grouped into various forms as following items.

- In case of appoingting type of upgrade Poller. For example, the type is 3536 and 3836 – "NodeType:3536", "NodeType:3836"
- In case of appointing resident such as PLoP Apartment Number 101 – "AptNum:0101"
- In the event of appointing ouprade Poller group. For example, the range of group is between PLoP Apartment Number 101 and PLoP Apartment Number 112 – "Range:AptNum:AptNum:0101:AptNum:0112"
- In case of appointing type for permission access to Apartment. For example, the type is 3536 as entrance system at PLoP Apartment Number 101 – "RangeAptNum:AptNum:0101:NodeType:3536"

Step 6 : Check Poller's status periodically.

When the server(Pusher) distributes the upgrade time though the scheduler, some of the clients are often off-line. Then the server waits for response from clients for a certain time. Because the client is off-line, an incorrect schedule is made which increase total upgrade time.

To solve this problem, Alive Check Manager that distinguishes whether the target Pollers(clients) are alive or not is needed as a separate upgrade module. Alive Check Manager sends cycle to the Pollers which are registered. Then those Pollers send the Alive message to the server according to the assigned cycle. For example, if the alive message does not come over the cycle * N times, it is necessary to change the state of Poller to dead and the Poller is removed from the upgrade target list.

Step 7 : Use the Timer or WatchDog to manage the time information from the pusher.

If Pusher and Poller have absolute time as TimeSpan, a complex time synchronization mechanism is needed. However, the interval of time is used as TimeSpan in the Half-Push/Half-Polling pattern. Instead of that, either Timer or Watcher(WatchDog)[4]is needed to control the time interval. In addition, current time and the time interval(TimeSpan) must be stored in a file or DB in case of system failure. So, we can know whether the upgrade request is needed when the system restarts. If the current time is greater than the time that we stored on the file or DB, the Poller rechecks whether the upgrade is possible to Pusher.

Step 8 : Consider appropriate File Transmission Mechanisms.

All systems have to use appropriate mechanisms depending on domain or situation. The optimal solution for every situation(Silver Bullet) does not exist. When you need to upgrade/patch many different kinds of devices, it is necessary to consider a variety of network environments. Either the Pollers which request mass file transfer may exist or the Pollers which periodically request small amounts of data may exist. To consider this situation, the File Transmission Strategy is chosen according to the type of Poller. To resolve these problems, "JAWS:A Framework for High-Performance Web Servers"[5] provides good solutions. In that study, the asynchronous transfer mechanism(Proactor)[8] like IOCP has bad performance for transferring small files.

9. KNOWN USES

– OMG CORBA Event Service
The Event service of RealTime CORBA it is not for upgrading, but the Event Channel method that replaces push method with pull(polling) method as data transmission method.

– Hybrid Push/Pull Download Model in Software Defined Radios
A Software-Defined Radio(SDR) system is a radio communication system where components that have typically been implemented in hardware(e.g. mixers, filters, amplifiers,

modulators/demodulators, detectors. etc.) are instead implemented using software on a personal computer or other embedded computing devices. When a new way of service starts, it is natural to replace existing terminals with new ones. However, the SDR performs mostly on Software instead of the existing semiconductor. Hybrid Push and Pull[12] which is one of the down models SDR offers is a good example of Half-Push/Half-Polling pattern.

– Samsung Homevita
Homevita, a home networking system from Samsung electronics, adopted the upgrade methods with a mixture of push and polling method. In the Homvita 1.0 version, it took the polling method for upgrade. However, it had big overhead by the request of many devices simultaneously. To avoid the problem, the system adopted the push upgrade method in the Homevita 2.0 version. After that, overhead was reduced. But the system needed a way to monitor dead devices. In the 2.5 version, a mixture of push and polling method are adopted. It reduces server overhead. And the devices which are alive keep the newest version.

10. RESULTING CONTEXT

The advantages of this pattern include:

- Solving the problem of heavy loads in very short periods of time caused by the Polling-based upgrade method and benefiting from the Push method which only upgrades specific clients.
- Reducing network traffic and load of the server/client because it doesn't need to check the version of the server periodically.
- Being able to upgrade specifically selected clients by grouping them.

Possible disadvantages are:

- It is not suitable for systems like vaccines that require all devices to be upgraded in case of emergency because the scheduling method used takes server load into account.
- It is impossible to upgrade clients if a problem occurs on the server(Pusher) while the program is running. Reliability must be guaranteed by copying Pusher components or using various fault tolerance methods[4].
- It is difficult to apply to a system such as P2P, where server and client information changes frequently.

11. RELATED PATTERNS

Publisher-Subscriber[7]

Also called the Observer pattern, it is used to synchronize the information between two components-Publisher and Subscriber. Copying the database from publisher to Subscriber can be a typical example. It is used when the pattern encounters a non-stop talker object, in other words, when overload occurs because of non-stop Polling.

Composite Message[1]

This pattern is used for marshaling/un-marshaling data, extending and adding messages you want to transfer while passing through layers. It is also used to create a transmission protocol for each device(Poller) in environments heterogeneous to the Half-Push/Half-Polling pattern. It is used in various distributed middleware.

Pipe & Filter[7]

This pattern is used when adding or filtering messages you want to transmit flexibly according to the circumstance, used internally in the aforementioned Composite Message. It is also used to filter unwanted data when building an Event Channel.

Broker[7]

This pattern removes direct dependency(location information, platform restrictions, etc.) between server and client. By delegating the location information of Pusher, which is in Poller, to Broker, Poller can remove itself of its direct dependency on the Pusher.

ACKNOWLEDGMENTS

We thank our shepherd Robert Hanmer for his valuable comments that contributed to improve this paper. The EVA (Pattern Evangelist Group), James Chang provided useful improvements and corrections.

REFERENCES

[1] Aamond Sane, Roy Campbell, "Composite Messages: A Structural Pattern for Communication between Components", OOPSLA'95 Workshop on Design Patterns for Concurrent, Distributed, and Parallel Object-Oriented Systems, 1995.

[2] F.Cristian, "Probabilistic Clock Synchronization", Distributed Computing, vol.3., pp.146-158, 1989.

[3] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, "Refactoring : Improving the Design of Existing Code", Addison-Wesley Professional, 1999

[4] Robert S. Hanmer, "WatchDog", Patterns for Fault Tolerant Software, John Wiley & Sons, October, 2007

[5] James C. Hu, Douglas Schmidt, "JAWS: A Framework for High-Performance Web Servers", Domain-Specific Application Frameworks: Frameworks Experience By Industry, John Wiley & Sons, October, 1999

[6] Michael Hicks, Jonathan T. Moore, Scott Nettles, "Dynamic Software Updating", ACM Transactions on Programming Laguages and Systems(TOPLAS), Voume 27, Issue 6

[7] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, "Pattern-Oriented Software Architecture Volume 1: A System of Patterns", John Wiley & Sons, 1996

[8] Douglas C. Schmidt, Michael Stal, Hans Rohert, and Frank Buschmann, "Pattern-Oriented Software Architecure Volum 2: Patterns for Concurrent and Networked Objects", ,John Wiley & Sons, 2000

[9] R. Gusella, S. Zatti: "The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3 BSD", IEEE Transactions on Software Engineering, Vol.15, July 1989

[10] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt, “The Design and Performance of a Real-time CORBA Event Service”, Proceedings of OOPSLA’97, Atlanta, Georgia, October, 1997

[11] Sameer Ajmani, Barbara Liskov, Liuba Shrira, “Scheduling and Simulation: How to Upgrade Distributed Systems”, In Proceedings of the 9th Workshop on Hot Topics in Operating Systems(HotOS IX)

[12] Jamadagni, Satish, Umesh M.N., “A PUSH download architecture for software defined radios”, 2000 IEEE international symposium on personal wireless communication