Towards a Self-Adaptive Middleware for Building Reliable Publish/Subscribe Systems

Sisi Duan[1], Jingtao Sun[2], and Sean Peisert[1]

[1]University of California, Davis, 1 Shields Ave, Davis CA, 95616, USA
[2]National Institute of Informatics, The Graduate University for Advanced Studies, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan
{sduan,speisert}@ucdavis.edu
sun@nii.ac.jp

Abstract. Traditional publish/subscribe (pub/sub) systems may fail or cause longer message latency and higher computing resource usage in the presence of changes in the execution environment. We present the design and implementation of Mimosa Pudica, an adaptive and reliable middleware for adapting various changes in pub/sub systems. At the heart of Mimosa Pudica are two design ideas. First, the brokers can elect leaders to manage the network topology in a distributed manner. Second, software components can be relocated among brokers according to the user's pre-defined rules. Through these two mechanisms, brokers can be connected in a self-adaptive manner to cope with failures and guarantee delivery of messages. In addition, brokers can effectively utilize their computing resources. Our experimental results of a large-scale pub/sub system show that in the presence of environmental changes, each self-adaptive process generates as few as 30 ms extra latency.

1 Introduction

Today's large-scale publish/subscribe (pub/sub) systems require dynamically applicability to be adaptive to various changes in systems and applications. For instance, in the presence of environmental changes, message loss and broker/link failures are desired to be handled. In addition, for many applications, the software components of an application may need to be migrated from one node to another, so as to be adaptive to limited computing resources and high loading at a node. However, most existing approaches propose solutions in the software layer while the pub/sub system structure itself is not able to be adaptive to frequent changes. We propose Mimosa Pudica, a middleware that is dynamically adaptive to various changes from both pub/sub systems and applications on top. Base on the middleware, we build a reliable pub/sub system and also improve the overall efficiency in system resource usage.

An amount of past research efforts have been devoted to developing reliable pub/sub systems. Most of them guarantee that messages will eventually be delivered. In order to guarantee message order in the presence of failures, previous efforts have relied heavily on the topology, either through redundant nodes or links. However, redundant nodes have a high cost in replication, and redundant links usually require brokers to store large amount of redundant information, which limits the scalability of a system and may even render brokers unusable.

In this paper, we propose a design of a self-adaptive and reliable pub/sub system that scales more efficiently by not requiring redundant nodes or storage. At the core of our system is Mimosa Pudica, a middleware that is adaptive to various changes. We employ two novel design ideas. First, brokers of pub/sub systems can elect leaders through our leader election algorithm to manage the rest of brokers in a distributed manner. Second, the leader can automatically relocate the software components between brokers to achieve dynamic adaptation of the pub/sub system, according to the user's pre-defined rules. Based on such a design, brokers can be dynamically added or deleted to handle failures. Furthermore, software components can be distributed to effectively utilize computing resources and to prevent from node failures.

We use distributed *destination databases* that can be accessed by the brokers to store routing information of brokers and all the pre-defined adaptation rules. In the presence of environmental changes, the brokers access the destination database to obtain a broker group information. After a leader election among the group, the leader compiles the adaptation rules and notify brokers the results. Different groups of brokers run independently in a distributed manner to adaptively manage topology and migrate software components. Through such a mechanism, the system can cope with failures and better utilize broker resources. In addition, due to the flexibility of our design, the software components of an application can be reused and the rules can be free assembled and reused for regular and repeated changes.

Our paper makes the following key contributions:

- We designed and implemented a middleware Mimosa Pudica. In the presence of environmental changes, the system self-adaptively manages the topology and relocates software components between brokers in a distributed manner.
- We implemented a reliable, crash-tolerant pub/sub system based on Mimosa Pudica. Our solution can be built on top of any existing topology. In addition, no redundancy of messages, brokers, or storage, is required.
- Our evaluation results show that each adaptation process only imposes a temporary of 30 ms to 50 ms extra latency to the event delivery, which proves the efficiency of our approach.

2 Related Work

Building reliable pub/sub systems have been widely studied [2, 3, 9–12, 26]. Periodic subscription [9], where subscribers actively re-issue their events [2], works well in preventing message loss. The use of redundant paths [2, 3, 10, 12] or redundant links [11] handles broker/link failures. As long as all the brokers in at least one path are correct, messages can be reliably delivered. However, it may consume high bandwidth and storage at brokers and become very inefficient in the absence of failures. P2S [3] on the other hand, demonstrates a framework of using existing fault-tolerant libraries in pub/sub systems. It directly adapts Paxos [14], a classic crash-tolerant replicated state machine approach. However, the current framework employs a centralized set of replicated brokers and must be carefully designed in scalable systems.

There are four types of self-adaptation mechanisms. The first type [18, 22] is policy-based. Most of them focus on how to define the context. The second type dynamically changes coordination between programs run on different computers [25]. It enables client-side objects to automatically select and invoke server-side objects according to the requirements and system architectures. However, this type only modifies the relationships between distributed programs instead of the computers executing them. The third type is genetic programming [13]. Most approaches focused only on target applications or systems such that they have no space to execute and evaluate large number of generated programs. The forth type is aspect-oriented programming (AOP) [23]. Unlike our work, existing adaptations do not support the migration of programs because reflective and AOP approaches are primitive to modify programs running on a single computer.

3 Approach

In this section we present background for our pub/sub system. We begin by introducing the preliminaries and then describe the design of *destination database*, the key component for data storage. Last, we show leader election, which is used to select a leader such that adaptation can be managed by brokers.

3.1 Preliminaries

We assume asynchronous model, where messages can be delayed, duplicated, dropped, or delivered out of order and brokers may crash and subsequently recover. For any n brokers between any pair of publisher and subscriber, up to $\lfloor \frac{n-1}{2} \rfloor$ crash failures are tolerated. In other words, in order to handle f broker failures, there are at least $2f + 1$ brokers on the path.

We aim to achieve the *in-order delivery*, where all the messages from a publisher to a set of corresponding subscribers are delivered in the same sequential order. Liveness guarantees that if a message is delivered to a subscriber, all the subscribers to the same topic eventually receive the same message. Liveness is ensured under *partial synchorny* [5]. That is, synchrony holds only after some unknown global stabilization time, but the bounds on communication and processing delays may be unknown.

3.2 Destination Database

We use a *destination database* that can be accessed by all the brokers. The destination database maintains all the routing information of the brokers and a set of pre-defined rules for adaptation purposes. When a broker communicates with the destination database and requests for group communication, the destination database replies with the identities of a group of brokers on the path based on the broker identity, the message information, and the corresponding publisher and subscriber information. It serves a simple purpose of storage, i.e., it does not manage the configurations of brokers or make any adaptation decisions. Instead, and all the adaptation decisions are made in a distributed manner by brokers.

In order to avoid single point of failure, we propose a two layer structure of distributing destination databases. The first layer contains replicated servers that stores metadata and the second layer contains several databases, each of which stores information of a set of brokers and a whole set of rules. The broker information can be replicated at different databases to prevent loss of data when certain database fails. When a broker requests for group information, it simply accesses the closest second layer database. The database replies directly if it has the information of all brokers on the path. Otherwise, it sends a request to the first layer database, obtains metadata, accesses the corresponding database(s) to get the information of the brokers, and sends a reply to the broker.

3.3 Leader Election

Leader election selects a leader among a set of brokers. A leader collects the information of environmental changes, makes decisions according to the adaptation rules as described in §4, and notifies all the brokers the adaptation decisions. We now describe the leader election process and illustrate it in Algorithm 1.

Algorithm 1 Leader Election Algorithm

```
1: **Initialization:**
2:   $B_i, B_j \cdots$                  {Brokers}
3:   $DD$               {Destination Database}
4:   $\mathit{\Delta}$                            {Timer}
5:   $v \leftarrow 0$                   {View Number}
6:   Leader()                 {Elect Leader}
7:   $timeout()$                    {Timeout}
8:   $starttimer()$             {Start Timer}
9:   $canceltimer()$           {Cancel Timer}
10:  $F()$              {Adaptation Results}
11: **Broker** $B_i$**:**
12: **on event** adaptation
13:   **send** $[\text{LE}, o, B_i, B_j, nd]$ **to** DD
14: **on event** $timeout(\mathit{\Delta})$
15:   $v \leftarrow v + 1$           {Re-Elect Leader}
16:   ElectLeader($v, group$)
17: **on event** $[\text{GI}, B_k \cdots B_p]$
18:   $group \leftarrow B_k \cdots B_p$      {Group Info}
19:   ElectLeader($v, group$)
20: **on event** ElectLeader($v, group$)
21:   $B_q \leftarrow$ Leader($group$)
22:   **send** $[\text{LEADER}, B_q, v]$ **to** $group$
23:   $starttimer(\mathit{\Delta})$              {Monitor}
24: **on event** $[\text{LEADER}, B_q, v]$
25:   $count \leftarrow count + 1$
26:   **if** $count \leftarrow f$ **and** $i \leftarrow q$
27:     $action \leftarrow$ F(rules)        {Actions}
28:     **send** $[\text{NL}, B_q, v, action]$ to $group$
29: **on event** $[\text{NL}, B_q, v, action]$
30:   $canceltimer(\mathit{\Delta})$
31: **Destination Database:**
32: **on event** $[\text{LE}, o, B_i, B_j, nd]$
33:   $group \leftarrow B_k \cdots B_p$           {Group}
34:   **send** $[\text{GI}, B_k \cdots B_p]$ **to** group
```

When a broker B_i (or publisher/subscriber in corner cases) requests for leader election, B_i sends a message $[\text{LE}, o, B_i, B_j, nd]$ to the destination database, where o represents the type of adaptation request, B_j is the broker to be added/deleted, and nd contains the corresponding information. For instance, if B_i detects B_j to be faulty, the message is $[\mathrm{LE}, 1, B_i, B_j, M(src, dst)]$, where 1 represents broker deletion, $M(src, dst)$ is the message B_i is currently forwarding from *src* to *dst*. The destination database then sends a message $[\mathrm{GI}, B_k \cdots B_p]$ to the brokers B_k to B_p between *src* and *dst*. After receiving the group information, the brokers start leader election. The leader election proceeds with views. All the brokers follow the same criteria when electing a new leader, as shown below. When the new leader receives at least $f + 1$ matching [LEADER] messages (including its own message), it sends a message to all the brokers to confirm its leadership and notifies brokers the adaptation results.

1) Broker B_q is elected such that a) B_q is on the path; b) B_q is not suspected to be faulty; c) B_q has not been elected in previous views; and d) B_q is the closest to the publisher on the path.
2) When a broker votes for a new leader, it starts a timer. If it has not received the [NL] message before its timer expires, it suspects the current leader to be faulty, increases v by 1 and votes for another new leader.

4 Design

This section describes the design of our Mimosa Pudica middleware system. We first present our system requirements and then describe the system architecture in details. We also show four adaptation rules and examples of applying the them to build our reliable pub/sub system.

4.1 Requirements

Existing middleware systems typically assume that formal descriptions focus on actions [24] and it is essential to identify which actions are controlled by the environment, which actions are controlled by the machine, and which actions of the environment are shared with the machine. Our Mimosa Pudica middleware focuses on where the software components should be migrated to and achieve the entire system's adaptability by relocating software components. Mimosa Pudica meets the following requirements.
Fault tolerance. Our middleware is designed to tolerate fail-stop broker/link failures (i.e., crashes) in a timely manner such that faulty brokers are removed and can be later recovered.
Self-adaptation. Distributed pub/sub systems essentially lack a global view due to the decoupling of publishers and subscribers. Our system coordinate software components between brokers in order to support their applications in a self-adaptive manner for higher efficiency in resource usage.
Separation of concerns. All the software components of an application should be defined independently with our adaptation mechanism. This is because the applications where adaptive rules are defined inside software components can not be reused. Both the software components and adaptive rules are desired to be reused for better resource usage.

Service availability. Our system guarantees that service should always be available with limited resources, whereas most existing approaches explicitly or implicitly assume that their targets of the systems have enriched resources.
General-purpose. Our adaptation mechanism is designed to be a practical middleware that also supports general-purpose applications in the system.

4.2 System Architecture

Our proposed approach dynamically adds/deletes brokers and deploys software components of an application from one broker to one or multiple brokers, according to the predefined rules. As a result, our distributed pub/sub system is self-adaptive to various changes.

At the core of our system is a middleware system between OS and applications, as shown in Fig. 1. This architecture consists of two important parts: an adaptation manager and a runtime system. The adaptation manager manages the runtime system. It controls the behavior of components, selects rules from destination database, and determines where and when to migrate the software components. The runtime system is responsible for managing, executing, and migrating software components, as well as enabling them to invoke methods at other software components. In order to use these methods during migration, the software components are first serialized and then migrate themselves from one server to another. When the software components arrive at their destinations, servers can communicate with each other for naming inspection.

Adaptation manager. In order to be self-adaptive to the changes of environmental properties, the deployment of components is managed by the adaptation manager. They are fully distributed and no centralized management server is required. In the presence of environmental changes, brokers follow several steps to be self-adaptive, as shown below.

Step 1: When a broker detects the environmental changes, it first send messages to the destination database to obtain the group information. The brokers select a leader according to leader election algorithm as shown in Algorithm 1.

Step 2: The leader invokes the adaptation rules, compiles them, and notifies brokers the adaptation results, e.g, which broker should be added/deleted, or which one or part of the software components should be migrated to other brokers.

Step 3: Depending on the adaptation rules and results, as described in §4.3, brokers activate different software

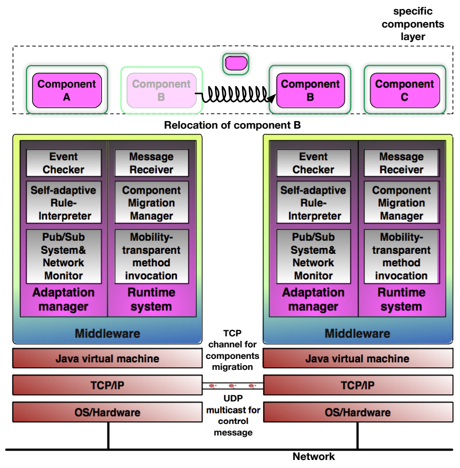

specific components layer
Component A
Component B
Component B
Component C
Relocation of component B
Event Checker
Message Receiver
Self-adaptive Rule-Interpreter
Component Migration Manager
Pub/Sub System& Network Monitor
Mobility-transparent method invocation
Adaptation manager
Runtime system
Middleware
Java virtual machine
TCP/IP
OS/Hardware
TCP channel for components migration
UDP multicast for control message
Network

Fig. 1. Mimosa Pudica middleware system architecture.

components. When a broker is deleted, neighbors of the broker are connected or new broker is added. The monitors of the brokers that are connected notify their software components. The brokers can then build the connection. On the other hand, when the software components are migrated to the destination broker, the monitor of destination broker notifies its software components. The methods of the migrated software component are then invoked by destination software components through reflection mechanism.

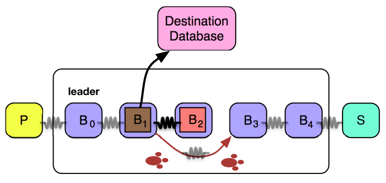
The adaptation manager contains three sub-modules: event checker, rule interpreter, and system and network monitor. The event checker identifies the type of event messages received by components runtime system and passes the event number to rule interpreter. The rule interpreter then searches rule from the destination database and executes it. Lastly, the system and network monitor dynamically monitors the state of brokers, e.g., threads count, CPU usage, used heap memory and the loaded class count, etc. Meanwhile, it also regularly monitors the changes of the component runtime system.

Component runtime system. The component runtime system has three modules: message receiver, component migration manager, and mobility-transparent method invocation. The message receiver, which has at most one message receiver thread, is responsible for receiving messages. The component migration manager receives command from adaptation manager. Each component has a particular life-cycle state. e.g., create, terminate, migrate, and duplicate. When the component state is changed, adaptation manager notifies the component migration manager the adaptation decision. The decision contains the components that should be moved, the components that should be cloned and moved, and the destination of migration. With this module, runtime systems at different servers can exchange messages through TCP channels by using Object Input/Output Stream. When a component is transferred over the network, both the code and the state of the component are transmitted into a bit stream and then transferred to the destination. At the destination side, the mobility-transparent method invocation module dynamically invokes the components through the class name and method name. The incomplete tasks will be run after migration.

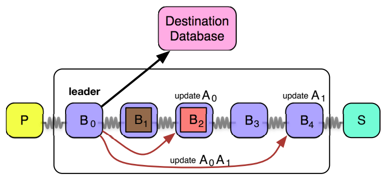
4.3 Adaptation Rules

When external environment changes, software components can be managed according to the predefined rules. To facilitate the definition of rules we use the Ponder language developed by the Imperial College[4]. Specifically, we use a subset of the Ponder language, i.e. the Ponder obligation rules. We list four rules using Ponder for topology management and software components mobility. We also include a few use cases of applying the rules in our pub/sub system. For simplicity, we illustrate the cases using a simple topology as shown in Fig. 2, where messages are sent and forwarded from publisher P to subscriber S through 5 brokers. In addition to the four rules, system developers can add new rules to destination database to meet different system requirements.

Rule 1 (Delete Brokers) Dynamically delete a number of brokers. By using this rule, system can reduce the number of the brokers and handle failures.



(a) Examples for Rule 1-3.



(b) Example for Rule 4.

Fig. 2. Examples of applying rules.

type oblig deleteBrokerRules(**target** database, Broker<T> broker){
subject AdaptationManager;
on deleteBrokerRequest();
do database.deleteBroker(broker);}

In the presence of failures. As illustrated in Fig. 2(a), broker B_2 crashes and its previous broker B_1 detects it. The leader B_0 compiles Rule 1 and deletes B_2. It notifies both B_1 and B_3. Broker B_1 and B_3 simply make a connection.

Rule 2 (Add Brokers) Dynamically add a number of brokers. The new broker only manages the routing information of its neighbors and is not required to know the state of other brokers. By using this rule, pub/sub system can better handle failures and improve system load balancing.

type oblig addBrokerRules(**target** database, Broker<T> broker){
subject AdaptationManager;
on addBrokerRequest();
do database.addBroker(broker);}

Too few brokers on a path. In the above example in Fig. 2(a), the leader B_0 can add a new broker B_5 to replace B_2. In this case, B_5 simply makes a connection with both B_1 and B_3 without knowing the identities of other brokers. Broker B_1, B_3, and B_5 then update their routing tables.

Rule 3 (Failure Judgment) Before the presence of broker failures, software components can be migrated to correct brokers to continue running. This rule works in systems where brokers are equipped with failure detectors or monitors. In this way, our system does not have to terminate the system operations.

type oblig failureJudgmentRules(**target** database, Broker<T> broker){
subject AdaptationManager;
on migrateBrokerRequest(Monitor, max_input_rate,; min_output_rate);
do database.goBroker(broker);
when max_input_rate <= min_output_rate;}

Before the presence of failures. When B_2 predicts its failure, it starts adaptation and sends message to the destination database. Leader B_0 compiles Rule 3 and migrates all the software components from B_2 to B_1, B_3, or both. In this specific case, broker B_1 and B_3 should also be connected for message delivery. After software migration, the leader also complies Rule 1 and connect B_1 and B_3.

Rule 4 (Task Transfer) Publishers may send different requests to brokers. However, some of the brokers may fail to communicate with subscribers. This rule can compress the parts of software component of brokers and transfer to one or several brokers. By using this rule, our system can effectively reduce the number of network transmission.

```
type oblig taskTransferRules(target database, Broker<List<T>>brokers){
subject    AdaptationManager;
on         transferBrokerRequest(Compression brokers, local_ip_info, remote_ip_info);
do         database.goBroker(brokers);
when       brokers.getBrokersID() <= User Defined;}
```

Broadcast to several brokers. As shown in Fig. 2(b), if B_0 receives an update command and is required to update two of the applications A_0 and A_1, B_0 will compress the two update commands and migrate to all the brokers that run at least one application, e.g., B_2 runs A_0 and B_4 runs A_1, B_0 migrates the update components to both B_2 and B_4. After receiving the update command, broker B_2 and B_4 retrieve the corresponding command and update A_0 and A_1 respectively.
Conflict Resolution. Adaptations may have conflicts with each other, even when each of them is appropriately composed. In our current implementation, all the rules are executed by the leader. Therefore, when there are conflicts between groups of brokers (e.g. overlapping brokers), the leaders of different groups first analyze whether there are conflicts between the rules of their visiting components. Once conflicts are found, the executing sequences are decided according to their arrival sequences. In other words, an adaptation request will be executed until all the conflicting requests that arrive earlier are executed. In the future, we will further develop the system such that each broker can simultaneously execute their rules by adding priorities or privileges to rule format [15].

5 Evaluation

In this section we evaluate the performance by assessing the adaptation latency in the presence of broker failures and software components migration. First, our approach handles broker failures by connecting neighboring brokers and introducing new brokers while no known previous work use similar approach. Second, the migration of software components prevents from failures and is shown to be very efficient. We carry out experiments on Deterlab [1], utilizing up to 30 machines. Each machine is equipped with a 3 GHz Xeon processor and 2 GB of RAM. They run Linux 2.6.12 and are connected through a 100 Mbps switched LAN. We use up to 24 publishers and subscribers. Publishers run concurrently with an average workload of $1,250$ events per second.

Implementation. Each component is implemented as a general-purpose and programmable entity. Defined as a collection of Java objects and packaged in the standard JAR file format, components can be migrated and duplicated between servers. Our middleware is built on the Java Virtual Machine (JVM) and can be abstracted away between different operating systems. The current implementation uses the Java object serialization package to marshal and duplicate components. The package dose not support the capture of stack frames of threads. Instead, when a component is duplicated, the runtime system issues events to invoke the specified methods. The methods are executed before the component is duplicated or migrated and active threads are suspended.

Adaptation Latency. We assess the adaptation delay of 1) adding/deleting brokers, as shown in Rule 1 and 2 in §4.3, and 2) migrating software components, as shown in Rule 3 and 4. We mainly evaluate two settings in the presence of broker failures: simple topology and bottleneck server crashes. Different sizes of random non-cyclic broker topologies are generated for each experiment. Simple topology simply evaluates failures in a single path where there is no side effect in the presence of broker failures. In comparison, the goal of the case where bottleneck broker crashes is to assess the latency when multiple paths request for adaptation in the presence of failures.

Add/Delete Brokers. We periodically inject random broker failures every 50 publications and assess end-to-end latencies. It can be observed in Fig. 3 that the average latency is 8 ms to 12 ms. When there are failures, subscribers experience a temporary 65 ms to 85 ms peak latency. The long latency resumes to normal after a few publications.

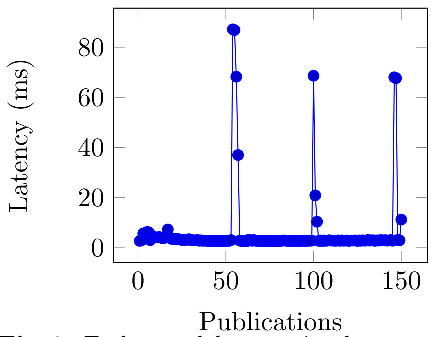


Fig. 3. End-to-end latency in the presence of broker failures.

We break down the peak latency into four phases: 1) timeout, where brokers use timers to detect the failures of their subsequent brokers; 2) vote for leader election, where brokers to obtain group information from destination database; 3) leader election, where brokers elect a new leader; and 4) adaptation, where the leader makes adaptation. We use instant acknowledgment (*ack*) messages for brokers to detect the failures, where if a broker has not received *ack* message before its timer expires after forwarding a message, it suspects its subsequent broker to be faulty.

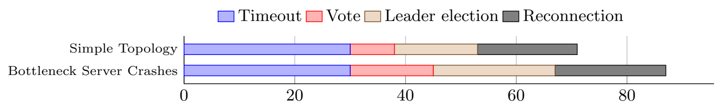


Fig. 4. Adaptation delay in details.

As observed in Fig. 4, the value of the timer is set to 30 ms, which is also the bottleneck of the overall delay. Indeed, if a smaller timer is used, the overall latency can be greatly reduced but it also increases the false negatives since slow brokers are detected to be faulty. The second phase generates 8 ms average latency for simple topology and 15 ms latency for complicated topology. This is due to the fact that paths with overlapping broker(s) are given access sequentially by destination database to avoid conflicts. In this particular experiment, the bottleneck server is the only overlapping server that crashes. Therefore, they run leader election concurrently, which generates 15 ms latency for simple topology and 22 ms in complicated topology. The adaptation phase causes 18 − 20 ms latency for both settings since leaders compile the rules independently.

Migrate Software Components. We assess the delay of software components migration. We run four applications, each of which corresponds to one pre-defined rule, to evaluate the performance. Each software component has a life-cycle, as shown in Table 1. When the requirements change, its life-cycle will be changed to another state. Our experiment results show that the four applications generate 161 ms, 201 ms, 189 ms, and 184 ms latencies respectively. The temporary delays of the four cases are small because we only migrate the source code and the state of the components. Among all the applications, "app.RemoteSearch" generates the longest delay. This is because all the corresponding threads and processes need to be deleted when executing the delete rule.

Table 1. Migration of software components.

Runtime_ID	Rule	Component_ID	Component_Name	Life_cycle	Component_Time	Delay(ms)
13618708105680001501261633499959	/Rules/AddRule	dc36fae696d04cd18ff1eab7429606f1	app.Chat	creation	5:32 PM	161
13618708105680001501261633499959	/Rules/DeleteRule	b89ebe96181540259ce8e09a4e858485	app.RemoteSearch	creation	5:35 PM	201
13618708105680001501261633499959	/Rules/FaiJudgmentRule	2ea7663f79fa479a8a974222caf353dc	app.FileTransfer	creation	5:37 PM	189
13618708105680001501261633499959	/Rules/UpdateRule	5c55518e36404973b6a62dc665b32c6c	app.Update	creation	5:40 PM	184
...	...	...	...	...	...	...

To summarize, a smaller value of the timers can reduce the overall latency but can also increase the false negatives. Also, when more than one overlapping brokers of multiple paths fail, the overall adaptation delay can also be increased.

6 Conclusion and Future Work

We present a self-adaptive middleware for building reliable pub/sub systems. Our approach does not require redundant brokers, network links, or storage at brokers in order to tolerate crash faulty brokers. It fits naturally in any existing topology. In addition, our approach self-adaptively manages the topology and software components among brokers and can be easily managed to serve different purposes. We have shown how our Mimosa Pudica middleware manages the adaptive rules in the presence of environmental changes. Our evaluation results show that our adaptation approach imposes a temporal period of slightly longer latency in the presence of environmental changes. In the future, we will further develop the system to address Byzantine failures and to add privileges for the adaptation rules and resolve the possible conflicts and divergences.

Acknowledgement

This research is based on work supported by the National Science Foundation under Grant Number CCF-1018871. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

References

1. T. Benzel. The science of cyber security experimentation: the DETER project. *ACSAC*, 2011.
2. R. Chand and P. Felber, "Xnet: A reliable content-based publish/subscribe system," in *SRDS*, 2004, pp. 264–273.

3. T. Chang, S. Duan, H. Meling, S. Peisert, and H. Zhang. P2S: a fault-tolerant publish/subscribe infrastructure. *DEBS*, 2014, pp. 189–197.
4. N. Damianou, N. Dulay, et al , “The Ponder Policy Specification Language,” in *POLICY*, 2001, pp.18–38.
5. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *JACM* 35(2):288–323, 1988.
6. P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, “The many faces of publish/subscribe,” *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003.
7. J. Hauer, et al. A component framework for content-based publish/subscribe in sensor networks. *EWSN*, pp. 369–385, 2008.
8. J. Hoffert, A. S. Gokhale, and D. C. Schmidt. Timely Autonomic Adaptation of Publish/Subscribe Middleware in Dynamic Environments. *IJARAS*, 2(4), pp. 1–24, 2011.
9. Z. Jerzak and C. Fetzer, “state in publish/subscribe,” in *DEBS*, 2009, pp. 1–12.
10. R. S. Kazemzadeh and H.-A. Jacobsen, “Reliable and highly available distributed publish/subscribe service,” in *SRDS*, 2009, pp. 41–50.
11. R. S. Kazemzadeh and H.-A. Jacobsen, “ Partition-Tolerant Distributed Publish/Subscribe Systems,” in *SRDS*, 2011, pp. 101–110.
12. R. S. Kazemzadeh and H.-A. Jacobsen, “Opportunistic multipath forwarding in content-based publish/subscribe overlays,” in *Middleware*, 2012, pp. 249–270, private communication.
13. J. R. Koza, “Genetic Programming,” On the Programming of Computers by Means of Natural Selection, MIT Press, 1992.
14. L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
15. E. Lupu, and M. Sloman, “Conflicts in policy-based distributed systems management,” , *IEEE Trans. on Software Engineering*, 25.6, 1999, pp.852–869.
16. P. Oreizy, N. Medvidovic, and R. N. Taylor “Runtime software adaptation: framework, approaches, and styles,” in *ICSE*, 2008, pp. 899–910, 2008.
17. T. Sivaharan, G. S. Blair, and G. Coulson , “Green: A configurable and reconfigurable publish-subscribe middleware for pervasive computing,” in *OTM Conferences*, 2005, pp. 732–749.
18. J. Sun and S. Ichiro, “Dynamic Deployment of Software Components for Self-Adaptive Distributed Systems,” in *IDCS*, 2014, LNCS 8729, pp. 149–203.
19. R. N. Taylor, N. Medvidovic, and P. Oreizy , “Architectural styles for runtime software adaptation,” in *WICSA/ECSA*, 2009, pp. 171–180.
20. M. A. Tariq, et al, “Dynamic publish/subscribe to meet subscriber-defined delay and bandwidth constraints,” in *Euro-Par*, 2010, pp. 458–470.
21. T. Yasuyuki, A. Ohsuga, and S. Honiden, “Rewriting Logic Model of Compositional Abstraction of Aspect-Oriented Software,” in *FOAL*, 2010, pp. 53–62.
22. T. Hiroki, et al, “A rule-based framework for managing context-aware services based on heterogeneous and distributed Web services,” in *SNPD*, 2014, pp. 1–6.
23. P. K. McKinley, S. M. Sadjadi, E. P. Kasten, et al, “Cheng: Composing Adaptive Software, ” in IEEE Computer Vol.37, No.7, 2004, pp.56-64.
24. P. Zave, and M. Jackson, “Four dark corners of requirements engineering,” *TOSEM*, 1997, pp.1–30.
25. J. Zhang and B. H. Cheng, “Model-based development of dynamically adaptive software,” in *ICSE*, 2006, pp. 371–380.
26. K. Zhang, V. Muthusamy, and H. Jacobsen, “Total order in content-based publish/subscribe systems,” in *ICDCS*, 2012.