Dynamic Self-Adaptation for Distributed Service-Oriented Transactions

Hassan Gomaa
Department of Computer Science
George Mason University
Fairfax, VA 22030
hgomaa@gmu.edu

Koji Hashimoto
Department of Computer Science
George Mason University
Fairfax, VA 22030
kojihashi@gmail.com

***Abstract*—Dynamic software adaptation addresses software systems that need to change their behavior during execution. To address reuse in dynamic software adaptation, software adaptation patterns, also referred to as software reconfiguration patterns, have been developed. A software adaptation pattern defines how a set of components that make up an architecture or design pattern dynamically cooperate to change the software configuration to a new configuration given a set of adaptation commands. This paper describes a dynamic self-adaptation pattern for distributed transaction management in service-oriented applications.**

Keywords-component; service-oriented architecture; dynamic software adaptation;software adaptation pattern;distributed transactions

I. INTRODUCTION

Dynamic software adaptation addresses software systems that need to change their behavior during execution. In self-managed and self-healing systems, systems need to monitor the environment and adapt their behavior in response to changes in the environment [10]. Adaptation can take many forms. It is possible to have a self-managed system which adapts the algorithm it executes based on changes it detects in the external environment. If these algorithms are pre-defined, then the system is adaptive but the software structure and architecture is fixed. However, if the adaptation necessitates changes to the software architecture, then the software architecture as well as the executable system will need to dynamically change at run-time. Rather than hand craft each case of dynamic adaptation of the software architecture, software adaptation can benefit from reuse in a similar way that designing software architectures has benefited from the reuse of software architectural and design patterns.

To address reuse in dynamic software adaptation, software adaptation patterns, also referred to as software reconfiguration patterns, have been developed. A software adaptation pattern defines how a set of components that make up an architecture or design pattern dynamically cooperate to change the software configuration to a new configuration given a set of adaptation commands. For each of the software architectural patterns [1,7, 8] that can be used to compose a software architecture, there is a corresponding software adaptation pattern, which defines how the software components and interconnections can be changed under predefined circumstances. Examples of software adaptation patterns are replacing one service with another in a client/service pattern and inserting a control component between two other control components in a distributed control pattern, etc. Previously developed adaptation patterns include the Master-Slave Adaptation Pattern, Centralized Control Adaptation Pattern, and Decentralized Control Adaptation Pattern [5, 6] and independent SOA service coordination [23].

The research described in this paper builds on software adaptation patterns [6, 23] and advances these concepts for service-oriented applications. In typical SOA applications, services are self-contained and loosely coupled, and orchestrated by coordination services. As there are many different types of service coordination, this paper considers dynamic adaptation based on SOA coordination patterns that capture the different kinds of coordination. In particular, this paper builds on previous work on dynamic adaptation of independent stateless coordination patterns to describe transaction-based distributed software adaptation.

This pattern involves the coordination of distributed stateful transactions in which updates to more than one service need to be coordinated, e.g., a transfer of electronic funds from one bank to another. In particular, a distributed transaction needs to have atomic operations involving updates to more than one service. With this approach, updates to the distributed services are coordinated such that they are either all performed (commit) or all rolled back (abort).This paper describes a dynamic self-adaptation pattern for distributed transaction management in service-oriented applications. This paper also describes validation of the adaptation pattern by means of a Web Service based prototype implementation of an Emergency Response System.

II. RELATED WORK

The dynamic adaptation patterns described in this paper address three research areas as follows: a) research into software architectural and design patterns [2,3,7] applied in particular to service-oriented architectures [16,17], b) research into dynamic reconfiguration and change management [6,9], and c) research into self-adaptive, self-managed or self-healing systems [4,10].

Dynamic software architectures and dynamic reconfiguration approaches have been applied to dynamically adapt software systems [4,10]. These approaches address incorporating dynamic reconfiguration into the architecture, as well as design and implementation of software systems for the purpose of run-time change and evolution. In [6,11,12] dynamic reconfiguration is applied to changing the configuration of a system from one configuration to another in a software product line while the system is operational. Research into self-adaptive, self-managed or self-healing systems [4,10] includes various approaches for monitoring the environment and adapting a system's behavior in order to support run-time adaptation.

Kramer and Magee [9,10] describe how a component must transition to a quiescent state before it can be removed or replaced in a dynamic software configuration. Ramirez et al. [13] describe applying adaptation design patterns to the design of an adaptive web server. The patterns include structural design patterns and reconfiguration patterns for removing and replacing components.

For service-oriented computing and service-oriented architectures, Li et al. [14] suggest the adaptable service connector model, so that services can be dynamically composed. Irmert et al. [15] provide a framework to adapt services at run-time without affecting application execution and service availability. Garlan and Schmerl [4] have proposed an adaptation framework for self-healing systems, which consists of monitoring, analysis/resolution, and adaptation.

In comparison with the previous approaches, this paper focuses on dynamic self-adaptation in service-oriented architectures. This paper describes a software adaptation pattern for transaction-based service coordination, in order to adapt not only services but also coordinator components.

III. SOFTWARE ADAPTATION FOR SOA

A. *Architecture Model for Software Adaptation*

The approach in this paper for software adaptation is compatible with the widely accepted three-layer reference architecture model for self-management [10]. This reference architecture for self-management originally comes from research [18,19] on control architectures for robotic systems. The architecture model consists of: 1) Goal Management layer—planning for change, 2) Change Management layer—manage the adaptation in response to changes in the environment reported from lower layer or in response to goal changes from above, and 3) Component Control layer—implements the run-time adaptation of the executing system in response to commands from Change Management.

B. *SASSY FRAMEWORK*

The software adaptation patterns described in this paper are developed as part of Self-Architecting Software Systems (SASSY), which is a model-driven framework for run-time self-architecting and re-architecting of distributed service-oriented software systems [20,21,25]. SASSY provides a uniform approach to automated composition, adaptation, and evolution of software systems. SASSY provides mechanisms for self-architecting and re-architecting that determine the best architecture for satisfying functional and Quality of Service (QoS) requirements [22,24].

In terms of the above three-layer reference architecture, adaptation patterns correspond to the component control layer of a self-managed system and they realize dynamic adaptation by adding or deleting components (and appropriate connectors if necessary) according to adaptation commands sent from the change management layer [23]. On the other hand, the goal management layer is in charge of producing change management plans (to satisfy QoS goals) that are executed at the change management layer. Thus, our focus in this paper is mainly on the component control layer for dynamic software adaptation.

C. *Software Coordination and Adaptation*

In SOA applications, services are intended to be self-contained and loosely coupled, so that dependencies between services are kept to a minimum. Instead of one service depending on another, it is desirable to provide coordination services (also referred to as coordinators) in situations where access to multiple services needs to be coordinated and/or sequenced. In SOA systems, loose coupling is ensured by separating the concerns of individual services from those of the coordinators, which sequence the access to the services. As there are many different types of service coordination, it is helpful to develop SOA coordination patterns to capture the different kinds of coordination. For each of these coordination patterns, there is a corresponding dynamic adaptation pattern [23].

D. *Independent Coordination Adaptation Pattern*

With this pattern, each coordinator instance operates independently of other coordinators in its interactions with services, as depicted in Figure 1. There are many instances of this coordinator, one for each client. Access to services can be sequential (e.g., airline reservation followed by hotel reservation), concurrent (e.g., contact multiple airline services concurrently), or some combination of sequential and concurrent (e.g., airline reservation followed by concurrent hotel and car reservations), as required by the application. Each interaction with a service is stateless and thus does not depend upon a previous interaction. Dynamic adaption for the independent coordination pattern was described previously in [23].

With the independent coordination adaptation pattern, a service can be replaced by another service when it has completed processing all outstanding service requests, either sequential or concurrent. When a service adaptation request is received, new service requests are queued until the dynamic service replacement is complete, and then forwarded to the new service. A coordinator object can be replaced when it has completed processing existing requests. In the meantime, new requests are queued until the dynamic coordinator replacement is complete, at which time queued service requests are sent to the new coordinator.

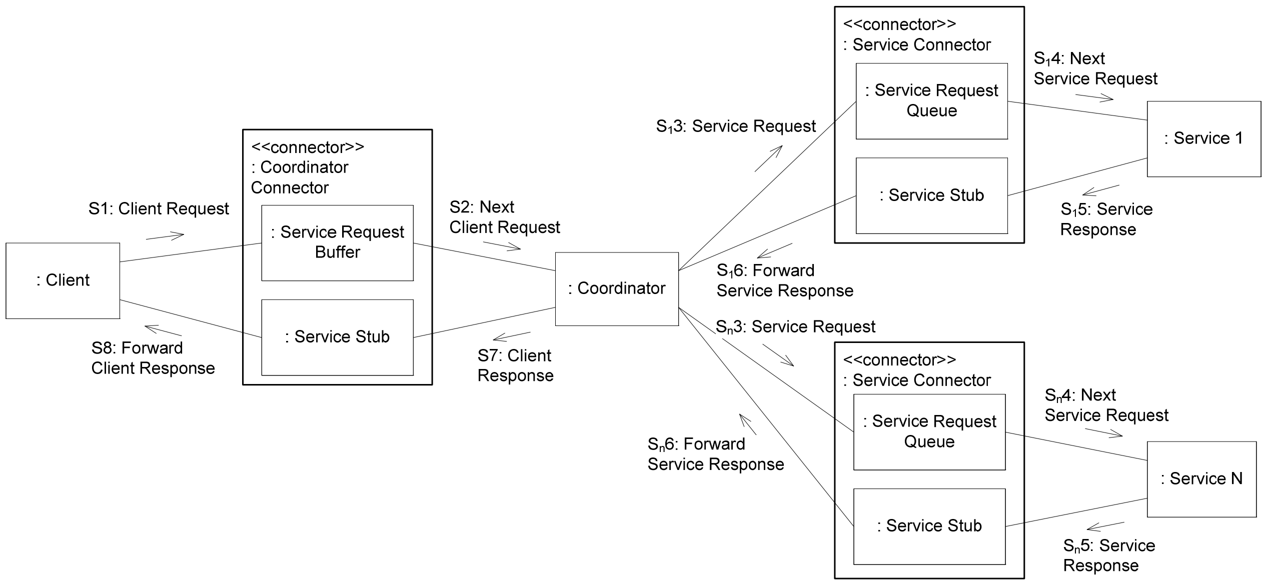


Figure 1: Independent service coordination

IV. Building Software Adaptation Patterns for SOA

A software adaptation pattern requires state- and scenario-based reconfiguration behavior models to provide for a systematic design approach. The adaptation patterns are described by adaptation interaction models (using communication or sequence diagrams) and adaptation state machine models [6,8].

A. *Software Adaptation State Machines*

An adaptation state machine defines the sequence of states a component goes through from a normal operational state to a quiescent state [9,23]. A component is in the Active state when it is engaged in its normal application computations. A component is in the Passive state when it is not currently engaged in a transaction it initiated, and will not initiate new transactions. A component transitions to the Quiescent state when it is no longer operational and its neighboring components no longer communicate with it. Once quiescent, the component is idle and can be removed from the configuration, so that it can be replaced with a different version of the component.

To enable adaptation patterns, as well as the corresponding code that realizes each pattern, to be more reusable, adaptation state machines are encapsulated in software adaptation connectors as discussed next.

B. *Software Adaptation Connectors*

Software adaptation connectors [23] are used to encapsulate adaptation state machine models so that adaptation patterns can be more reusable. The goal of an adaptation connector is to separate the concerns of an individual service from dynamic adaptation, i.e., the adaptation connector implements the adaptation mechanism for its corresponding service.

The adaptation patterns described in this paper include two different types of adaptation connectors, *coordinator connector* and *service connector*, as shown in Fig. 1. A service connector behaves as a proxy for a service, such that its clients can interact with the connector as if it was the service. The service connector implements the adaptation mechanism for its corresponding service, including the interaction with the change management layer (see next section) and the management of the operational states of the service. The adaptation state machine for a given adaptation pattern is encapsulated in the corresponding adaptation connector.

V. Transaction-Based Self-Adaptation Pattern

This section describes in detail the self-adaptation of services participating in atomic transactions. In distributed systems including service-oriented systems, a **transaction** consists of two or more operations that perform a single logical function that must be completed in its entirety or not at all [27]. For transactions that need to be atomic (i.e., indivisible), services are needed to begin the transaction, commit the transaction, or abort the transaction. In a service-oriented system, transactions are typically used for updates to distributed services that need to be atomic—for example, transferring funds from an account at one bank to an account at a different bank.

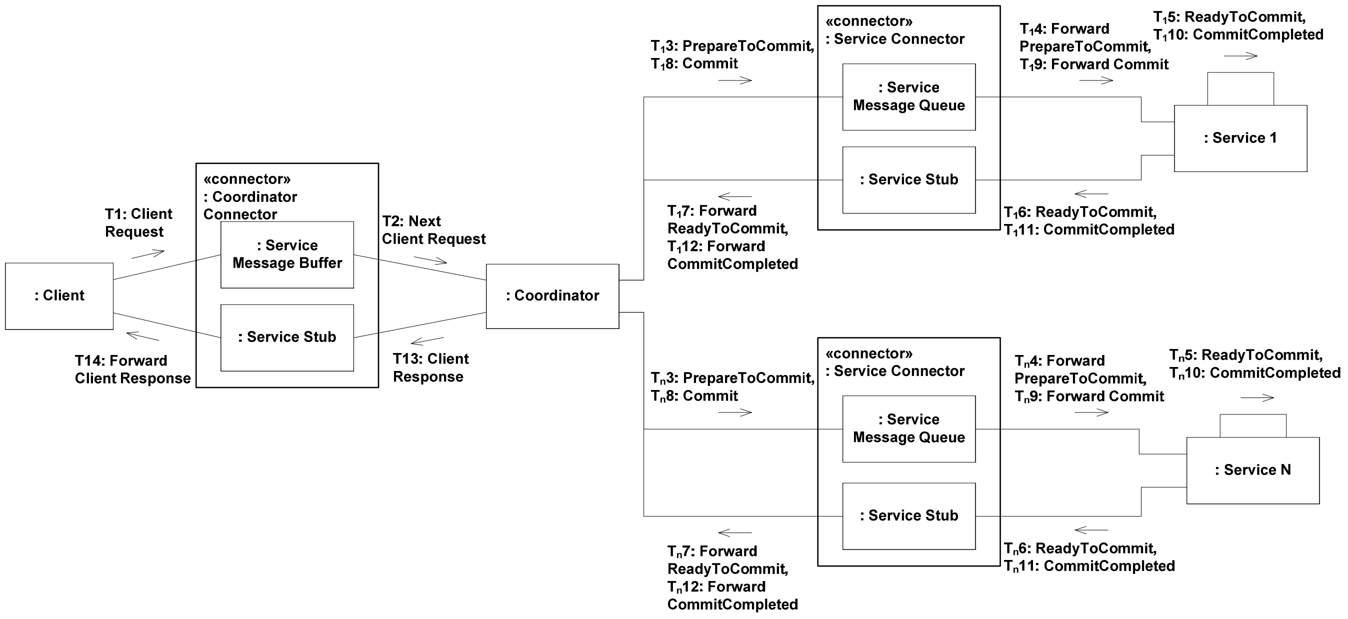
A. *Two-Phase Commit Protocol*

The Two-Phase Commit protocol is a well-known approach for managing atomic transactions in distributed systems [27]. This section describes a software adaptation pattern for services whose updates are coordinated using the Two-Phase Commit protocol.

Consider the self-adaptation pattern for Two-Phase Commit Coordination. The Two-Phase Commit Protocol synchronizes updates on different nodes in a distributed application. The result of the Two-Phase Commit Protocol is that either a transaction is committed (in which case all updates to distributed services succeed) or the transaction is aborted (in which case all updates to distributed services fail). In the case of a transfer transaction between two bank accounts maintained at two separate bank servers, the transfer transaction consists of two operations – a debit operation at one bank and a credit operation at the second bank. Since the transfer transaction is atomic, it must be either committed (both credit and debit operations occur) or aborted (neither the credit nor the debit operation occurs).

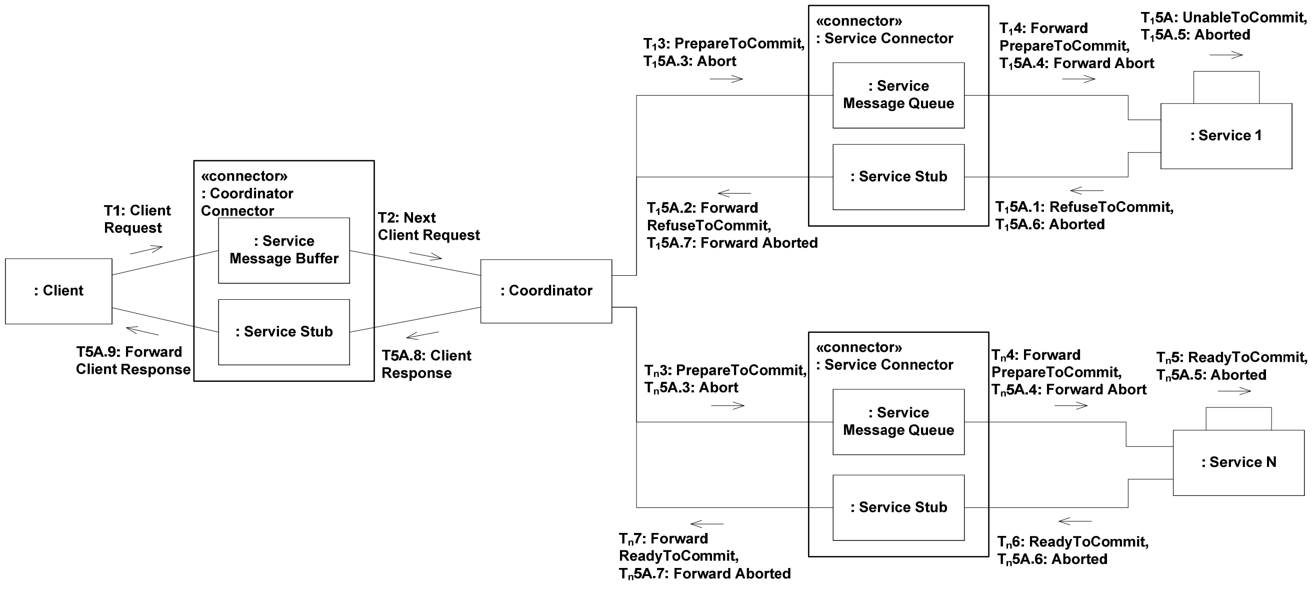
B. Service Update Coordination in Two-Phase Commit Pattern

Consider a general case where a coordinator executes the Two-Phase Commit Protocol with N participants (services). The communication diagram depicts the message exchange sequences in the cases of the transaction committed and aborted in Figures 2 and 3 respectively.



: Client
T1: Client Request
T14: Forward Client Response
«connector» : Coordinator Connector
: Service Message Buffer
: Service Stub
T2: Next Client Request
T13: Client Response
: Coordinator
$T_1$3: PrepareToCommit, $T_1$8: Commit
$T_1$7: Forward ReadyToCommit, $T_1$12: Forward CommitCompleted
«connector» : Service Connector
: Service Message Queue
: Service Stub
$T_1$4: Forward PrepareToCommit, $T_1$9: Forward Commit
$T_1$6: ReadyToCommit, $T_1$11: CommitCompleted
$T_1$5: ReadyToCommit, $T_1$10: CommitCompleted
: Service 1
T_n3: PrepareToCommit, T_n8: Commit
T_n7: Forward ReadyToCommit, T_n12: Forward CommitCompleted
«connector» : Service Connector
: Service Message Queue
: Service Stub
T_n4: Forward PrepareToCommit, T_n9: Forward Commit
T_n6: ReadyToCommit, T_n11: CommitCompleted
T_n5: ReadyToCommit, T_n10: CommitCompleted
: Service N

Figure 2: Two-Phase Commit coordination communication diagram: committed case



: Client
T1: Client Request
T5A.9: Forward Client Response
«connector» : Coordinator Connector
: Service Message Buffer
: Service Stub
T2: Next Client Request
T5A.8: Client Response
: Coordinator
$T_1$3: PrepareToCommit, $T_1$5A.3: Abort
$T_1$5A.2: Forward RefuseToCommit, $T_1$5A.7: Forward Aborted
«connector» : Service Connector
: Service Message Queue
: Service Stub
$T_1$4: Forward PrepareToCommit, $T_1$5A.4: Forward Abort
$T_1$5A.1: RefuseToCommit, $T_1$5A.6: Aborted
$T_1$5A: UnableToCommit, $T_1$5A.5: Aborted
: Service 1
T_n3: PrepareToCommit, T_n5A.3: Abort
T_n7: Forward ReadyToCommit, T_n5A.7: Forward Aborted
«connector» : Service Connector
: Service Message Queue
: Service Stub
T_n4: Forward PrepareToCommit, T_n5A.4: Forward Abort
T_n6: ReadyToCommit, T_n5A.6: Aborted
T_n5: ReadyToCommit, T_n5A.5: Aborted
: Service N

Figure 3: Two-Phase Commit coordination communication diagram: aborted case

Once the Coordinator receives a client request (message T2 in Fig. 2), it starts the Two-Phase Commit Protocol. In the first phase, the coordinator sends a "T_x3: PrepareToCommit" message to each participant service. Each participant service locks the record, performs the transaction, and then sends a "T_x6: ReadyToCommit" message to the coordinator (Fig. 2). If a participant service is unable to perform the transaction, it sends a "$T_x5A.1$: RefuseToCommit" message (Fig. 3). The coordinator waits to receive responses from all participants.

When all participant services have responded, the coordinator proceeds to the second phase of the Two-Phase Commit Protocol. If all participants have sent "T_x6: ReadyToCommit" messages, the coordinator sends the "Commit" message (T_x8 in Fig. 2) to each participant service. Each participant service makes the transaction permanent, unlocks the record, and sends a "T_x11: CommitCompleted" message to the coordinator. The coordinator waits for all "CommitCompleted" messages. The coordinator sends a response to its client (T13) after all the messages have been received.

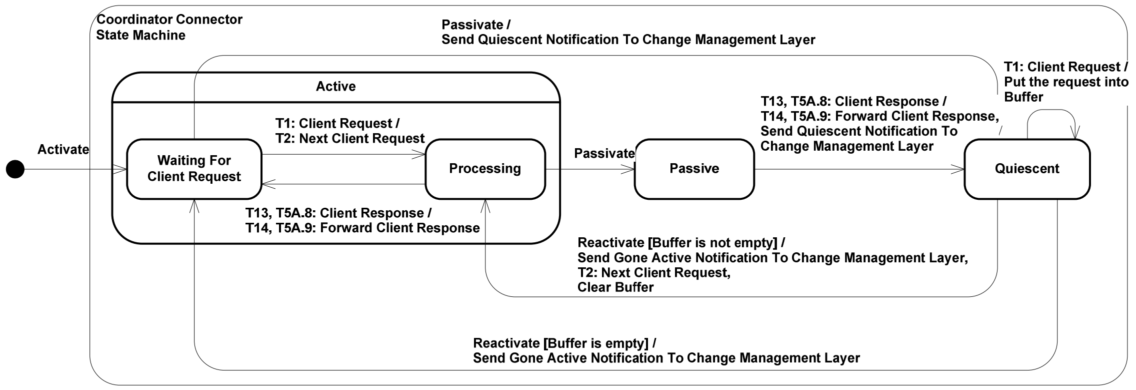
If a participant service responds to the "PrepareToCommit" message with a "ReadyToCommit" message, it is committed to completing the transaction. The participant service must then complete the transaction even if a delay occurs (e.g., even if it goes down after it has sent the "ReadyToCommit" message). If, on the other hand, any participant service responds to the "PrepareToCommit" message with a "RefuseToCommit" message, the coordinator sends a "$T_x5A.3$: Abort" message to all participants (Fig. 3). The participants then roll back the transaction.

Based on the Two-Phase Commit Protocol described above, the coordinator component can only be removed or replaced after it has received "CommitCompleted" or "Aborted" message from all the participant services and sent its response to the client. On the other hand, a participant service can be removed or replaced after it completes the current transaction of the protocol in the case of a sequential service, or after completing the current set of transactions (from different clients) in the case of a concurrent service.

C. *State Machine for Dynamic Self-Adaptation of Coordinator*

Next, consider the dynamic adaptation of the coordinator for the two-phase commit protocol. As described earlier, the adaption sequence for the coordinator is captured in an adaptation state machine, which is encapsulated in the connector for the coordinator. It is not possible to change the coordinator while it is participating in a two-phase commit transaction. Adaptation can only be carried out after it has completed a transaction and before it starts the next transaction.

As described in Section IV, the coordinator connector encapsulates and executes the adaptation state machine for the coordinator, shown in Figure 4. (Because of this, the state names reflect the states of the coordinator and not the connector). There are three main states, Active, Passive, and Quiescent. In the Active state, the coordinator is operating normally and its state machine is in one of the two substates of the composite Active state. The initial substate is Waiting for Client Request. When the client sends a Client Request (message T1 in Fig 2), the statechart transitions to Processing substate (event T1 in Fig 4) and forwards the next client request to the Coordinator (action T2 in Fig 4 and corresponding outgoing message in Fig 2). The Coordinator then starts carrying out the phases of the two phase commit protocol while the connector remains in the Processing substate. When the coordinator receives a message from the last service to respond with a Commit Completed message (T_x12 in Fig. 2) or Aborted ($T_x5A.7$ in Fig. 3), it sends the Client Response to the connector (T13 in the commit case or T5A.8 in the abort case). In either case, the state machine transitions back to the Waiting for Client Request substate and the action results in the Coordinator Connector forwarding the response to the client (T14 for commit or T5A.9 for abort).



Coordinator Connector State Machine
Passivate / Send Quiescent Notification To Change Management Layer
Active
T1: Client Request / T2: Next Client Request
Activate
Waiting For Client Request
Processing
T13, T5A.8: Client Response / T14, T5A.9: Forward Client Response
Passivate
Passive
T13, T5A.8: Client Response / T14, T5A.9: Forward Client Response, Send Quiescent Notification To Change Management Layer
Quiescent
T1: Client Request / Put the request into Buffer
Reactivate [Buffer is not empty] / Send Gone Active Notification To Change Management Layer, T2: Next Client Request, Clear Buffer
Reactivate [Buffer is empty] / Send Gone Active Notification To Change Management Layer

Figure 4: Coordinator connector state machine

The connector initiates the dynamic adaptation of the coordinator when the change management (CM) layer sends it the Passivate adaptation command. If the connector is in the Waiting for Client substate (Fig 4) when it receives a Passivate command, it is in fact idle and can therefore transition directly to the Quiescent state; it also sends a quiescent notification message to CM. If the connector is in the Processing substate when it receives a Passivate command, it transitions to the Passive state, which is an intermediate state where the coordinator is still interacting with the services to complete the transaction.

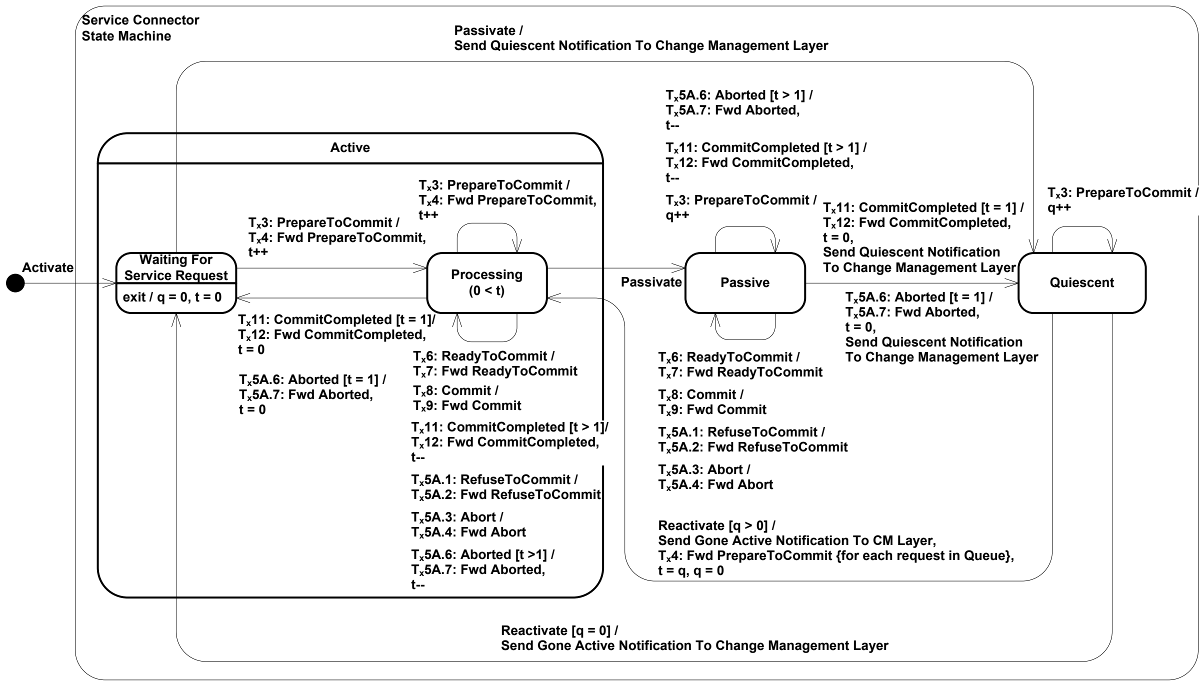
When the coordinator receives a response from the last service, it sends the client response to the connector (T13 in the commit case or T5A.8 in the abort case). The connector then transitions to the Quiescent state; the actions are to forward the response to the client (T14 or T5A.9), and to send a quiescent notification to CM.

While in the quiescent state, the connector could receive a new request from the client, which it puts into a request buffer. When the connector receives a Reactivate command from CM, it transitions to either the Waiting For Client Request or the Processing substate depending on whether or not there is a client request in the request buffer. If there is a client request, the request is removed from the buffer and sent to the coordinator, and the state machine transitions to the Processing substate. On the other hand, if the request buffer is empty, the state machine transitions to Waiting For Client Request. In either case, the connector notifies CM that the coordinator has returned to Active state.

D. *State Machine for Dynamic Self-Adaptation of Service*

Next, consider the adaption of services that participate in the two-phase commit pattern. As before, the adaptation state machine for the service is encapsulated in the service connector. Fig. 5 depicts the adaptation state machine executed by the service connector in the case of a concurrent service that has multiple threads processing client requests. The state machine is for a concurrent service that can handle multiple client transactions concurrently from different clients. A count t is used to keep track of the number of currently executing concurrent transactions. The state machine connector monitors its corresponding service to determine if it is participating in transactions of the Two-Phase Commit Protocol. While participating in one or more atomic transactions, a service cannot be adapted. If adaptation is required, new atomic transaction requests are queued up, while existing transactions are processed to completion. Once they have completed, the service adaptation is carried out and then queued transactions are sent to the new service.

If a "PrepareToCommit" message arrives in the "Waiting For Service Request" state, it is immediately forwarded to the service and the state machine transitions to Processing state. For subsequent messages, the connector stays in the "Processing" state, and forwards intermediate messages such as "ReadyToCommit", "Commit", "RefuseToCommit", and "Abort" to the service. The state machine keeps track of the number of transactions currently executed by the service by incrementing the variable t for each new transaction (arrival of Tx3Prepare to Commit message) and decrementing it as each transaction completes



Service Connector State Machine
Passivate / Send Quiescent Notification To Change Management Layer
Active
Activate
Waiting For Service Request
exit / q = 0, t = 0
Processing (0 < t)
Passive
Quiescent
Passivate
Tx3: PrepareToCommit / Tx4: Fwd PrepareToCommit, t++
Tx3: PrepareToCommit / Tx4: Fwd PrepareToCommit, t++
Tx11: CommitCompleted [t = 1]/ Tx12: Fwd CommitCompleted, t = 0
Tx5A.6: Aborted [t = 1] / Tx5A.7: Fwd Aborted, t = 0
Tx6: ReadyToCommit / Tx7: Fwd ReadyToCommit
Tx8: Commit / Tx9: Fwd Commit
Tx11: CommitCompleted [t > 1]/ Tx12: Fwd CommitCompleted, t--
Tx5A.1: RefuseToCommit / Tx5A.2: Fwd RefuseToCommit
Tx5A.3: Abort / Tx5A.4: Fwd Abort
Tx5A.6: Aborted [t >1] / Tx5A.7: Fwd Aborted, t--
Tx5A.6: Aborted [t > 1] / Tx5A.7: Fwd Aborted, t--
Tx11: CommitCompleted [t > 1] / Tx12: Fwd CommitCompleted, t--
Tx3: PrepareToCommit / q++
Tx11: CommitCompleted [t = 1] / Tx12: Fwd CommitCompleted, t = 0, Send Quiescent Notification To Change Management Layer
Tx3: PrepareToCommit / q++
Tx5A.6: Aborted [t = 1] / Tx5A.7: Fwd Aborted, t = 0, Send Quiescent Notification To Change Management Layer
Tx6: ReadyToCommit / Tx7: Fwd ReadyToCommit
Tx8: Commit / Tx9: Fwd Commit
Tx5A.1: RefuseToCommit / Tx5A.2: Fwd RefuseToCommit
Tx5A.3: Abort / Tx5A.4: Fwd Abort
Reactivate [q > 0] / Send Gone Active Notification To CM Layer, Tx4: Fwd PrepareToCommit {for each request in Queue}, t = q, q = 0
Reactivate [q = 0] / Send Gone Active Notification To Change Management Layer

Figure 5: Service connector state machine for a concurrent service

(arrival of either Tx11 Commit Completed or Tx5A.6 Aborted messages). If this is the last executing transaction, then the state machine returns to Waiting for Service Request state. Note that even if the state machine receives "CommitCompleted" or "Aborted" messages, it will remain in the "Processing" state if t > 1,since the service is still processing other transactions.

If the connector receives a Passivate command from CM, the connector transitions to the "Passive" state. In the "Passive" state, the connector queues new transactions (i.e., "PrepareToCommit" messages) on the request queue, since it will not initiate new transactions in this state but will continue to process existing transactions. Thus, while in the Passive state, the connector forwards intermediate messages of existing transactions to the service. The connector remains in the "Passive" state as long as there are transactions to process (t > 1) even when it receives "CommitCompleted" or "Aborted" messages. On the other hand, when the connector receives a "CommitCompleted" or "Aborted" message for the last executing transaction (t=1), it transitions to the "Quiescent" state. In this state, the service can be replaced because the service is no longer executing atomic transactions.

If further transaction requests are received in the "Quiescent" state, they are added to the new request queue and the queue count is incremented. When a Reactivate command is sent by CM, the state machine transitions out of "Quiescent" state and it notifies CM that it is now Active. The state machine transitions to Waiting for Service Request if the new request queue is empty. Otherwise, it transitions to Processing state and forwards all outstanding transaction requests (i.e., Prepare to Commit requests) in the transaction queue to the new service.

VI. IMPLEMENTATIONOF DYNAMIC SOFTWARE ADAPTATION FRAMEWORK

The SOA adaptation patterns were implemented as part of the SASSY dynamic software adaptation framework shown in Figure 6.The SASSY framework was developed using open-source SOA frameworks, Eclipse Swordfish and Apache CXF. Eclipse Swordfish is an open-source, extensible ESB (Enterprise Service Bus), built upon Apache ServiceMix. The prototype software adaptation framework was developed on top of Swordfish which is based on OSGi so that components can be integrated into Swordfish at runtime. In the framework prototype, the goal management and change management layers are implemented and integrated into Swordfish framework as service components. In addition, SOA coordinators are deployed on Swordfish at runtime as components.

Apache CXF is an open-source web-services framework which supports standard APIs such as JAX-WS and JAX-RS as well as WS standards including SOAP, WSDL, WS-Addressing, WS-Policy, etc. In the emergency response system example, the Building Locator, Occupancy Awareness, and Fire Station services are implemented on top of Apache CXF as if they are external services.

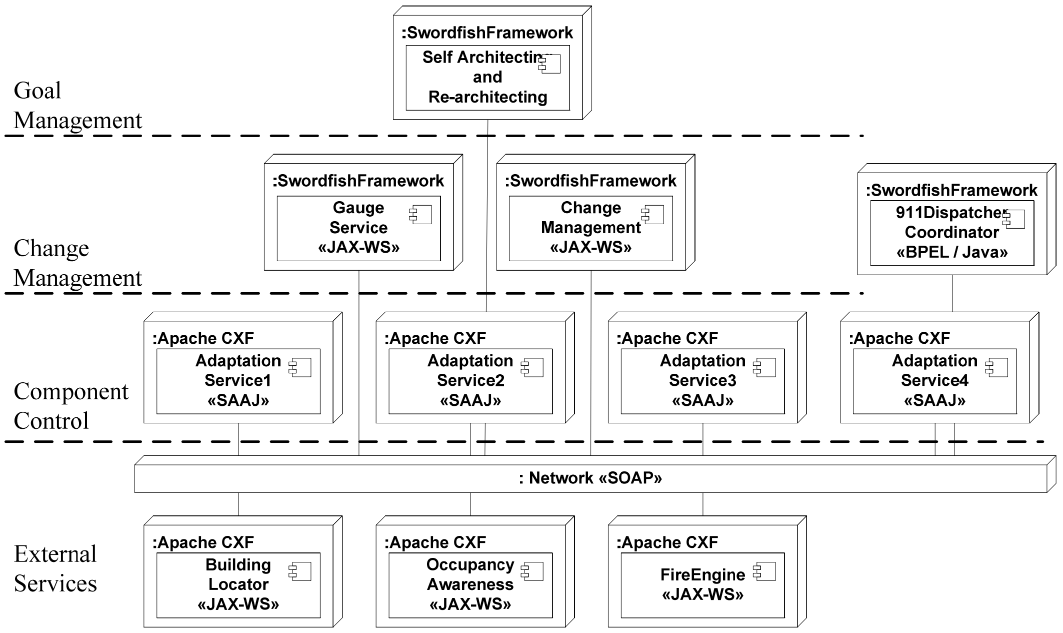


Figure 6: SASSY prototype dynamic software adaptation framework

Unlike other SASSY services, the adaptation connector was implemented as an independent web service on top of Apache CXF instead of Swordfish, because Swordfish, as an ESB, doesn't allow a component to dynamically change the service (provider) it invokes. The adaptation connector was implemented using SAAJ API, a low-level Web-Service API.

The implementation utilizes a SOAP header attached to a message indicating which kind of message it is, as follows:

1) SOAP header element "beginTransaction" in a request message indicates that the message initiates a new transaction. Its response is to be intermediate.

2) SOAP header element "intermediateTransaction" in a request message indicates that the message is intermediate. Its response is also to be intermediate.

3) SOAP header element "endTransaction" in a request message indicates that its response finishes the transaction.

4) If no SOAP header elements are attached to a request message, the message is considered to be for a stateless service.

In the two-phase commit coordination adaptation pattern, SOAP headers are attached to messages received by a service connector as follows:

- "T_x3: PrepareToCommit": 1) SOAP header element "beginTransaction" is attached.
- "T_x8: Commit": 3) SOAP header element "endTransaction" is attached.

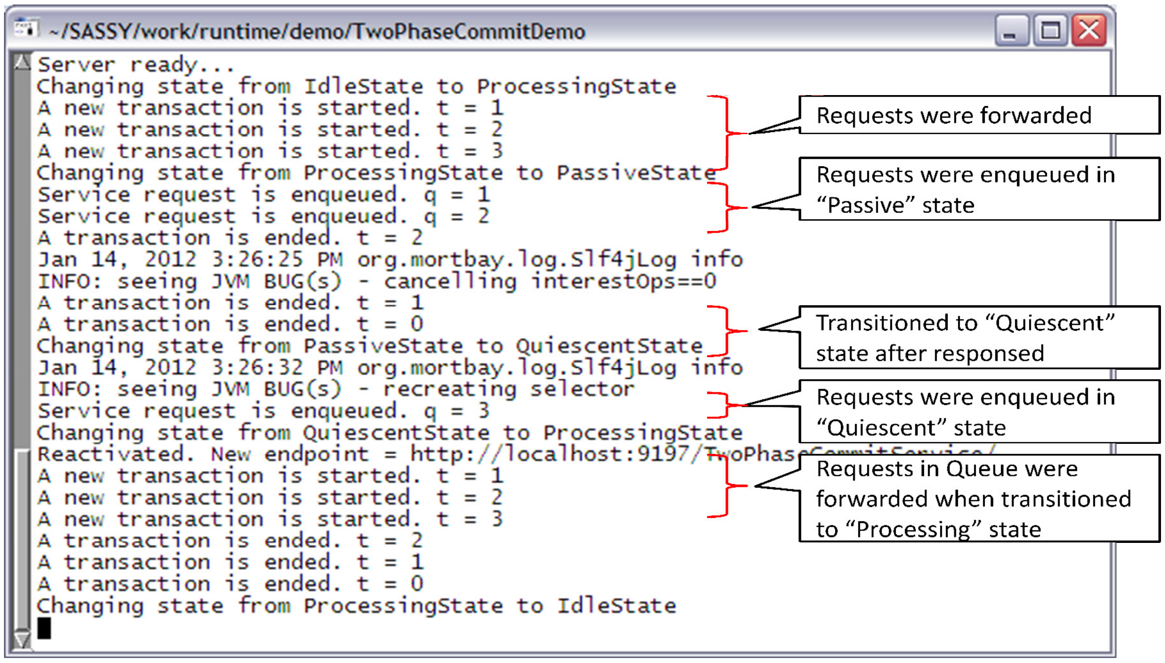
VII. Validation of SOA Adaptation Patterns

The SOA adaptation patterns were validated using the prototype implementation of the software adaptation framework described above. The validation consists of executing the change management scenario, performing the software adaptation from one configuration to another, and resuming the application after the adaptation.

This section describes the validation of the two-phase commit coordination adaptation pattern. A prototype implementation of Two-phase Commit Protocol involved the services randomly sending either "ReadyToCommit" or "RefuseToCommit" messages in response to a "PrepareToCommit" request message. The validation consisted of several runs in which the service connector and coordinator connectors were sent commands to passivate by the change management layer while processing atomic concurrent transactions. For each run, the connectors would output traces to depict the transitions through the states of the adaptation state machine.

This implementation executed the adaptation state machines in Figures 4 and 5 for the Emergency Response System consisting of three services. The executions trace of the service connector is shown in Fig 7.

The execution trace shows that the state machine, while in Processing state, initiated three transactions (through the Prepare to Commit message sent to the service). The service connector then received a Passivate command resulting in it transitioning to Passive state. Two further requests to Prepare to Commit arrived, which were queued in the


~/SASSY/work/runtime/demo/TwoPhaseCommitDemo
Requests were forwarded
Requests were enqueued in "Passive" state
Transitioned to "Quiescent" state after responsed
Requests were enqueued in "Quiescent" state
Requests in Queue were forwarded when transitioned to "Processing" state

```
Server ready...
Changing state from IdleState to ProcessingState
A new transaction is started. t = 1
A new transaction is started. t = 2
A new transaction is started. t = 3
Changing state from ProcessingState to PassiveState
Service request is enqueued. q = 1
Service request is enqueued. q = 2
A transaction is ended. t = 2
Jan 14, 2012 3:26:25 PM org.mortbay.log.Slf4jLog info
INFO: seeing JVM BUG(s) - cancelling interestOps==0
A transaction is ended. t = 1
A transaction is ended. t = 0
Changing state from PassiveState to QuiescentState
Jan 14, 2012 3:26:32 PM org.mortbay.log.Slf4jLog info
INFO: seeing JVM BUG(s) - recreating selector
Service request is enqueued. q = 3
Changing state from QuiescentState to ProcessingState
Reactivated. New endpoint = http://localhost:9197/TwoPhaseCommitService/
A new transaction is started. t = 1
A new transaction is started. t = 2
A new transaction is started. t = 3
A transaction is ended. t = 2
A transaction is ended. t = 1
A transaction is ended. t = 0
Changing state from ProcessingState to IdleState
```

Figure 7: Execution trace for Service Connector in two phase commit pattern

request queue. The three original transactions gradually terminate with the arrival of "CommitCompleted" messages. After the third transaction has terminated, the connector state machine transitioned to Quiescent state, at which time the service is replaced. While in Quiescent state, a further Prepare to Commit request arrives, which is also queued. After the service has been replaced, the connector receives a Reactivate command. As it has three outstanding transaction requests in the queue, the state machine will transition from Quiescent state to Processing state and send the three Prepare to Commit messages to the new service. The event sequence in the event trace just described is faithful to the service connector state machine given in Figure 5.

The execution trace of the coordinator connector (Fig 8) shows a transaction being initiated followed by a Passivate command which results in the coordinator connector transitioning from Processing state to Passive state. When the transaction ends, the state machine transitions to the Quiescent state, in which the coordinator is replaced. Another transaction (service request) message arrives, which is queued as the state is Quiescent. After replacing the coordinator, CM sends a Reactivate message to the connector. The state machine then transitions to Processing state and sends the queued transaction to the coordinator. Thus, the event sequence in the event trace is faithful to the coordinator connector state machine given in Figure 4.

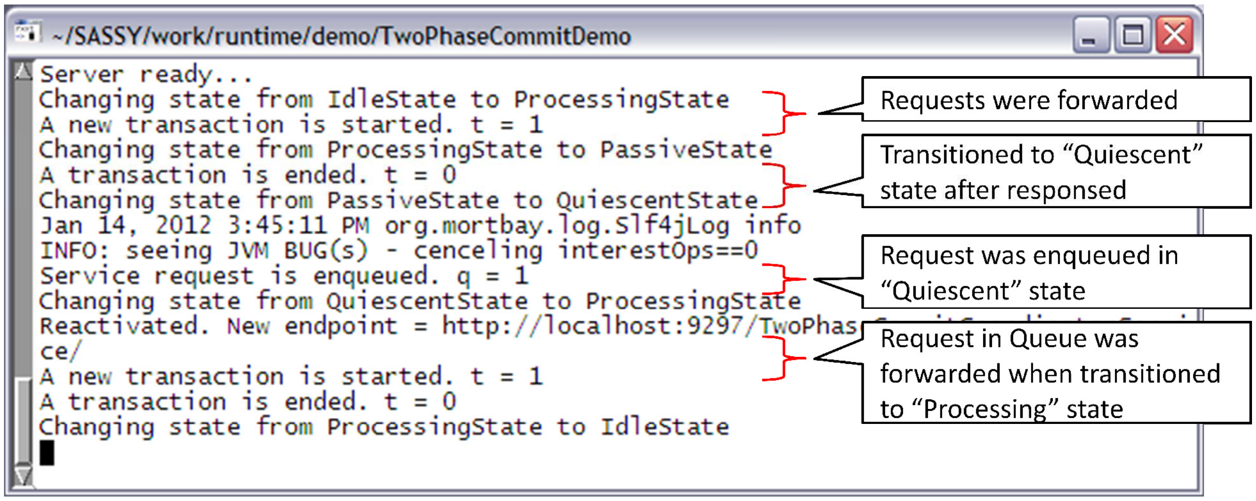
VIII. Discussion

This paper has described the dynamic transaction-based self-adaptation pattern, which involves adaptation of a service or adaptation of a coordinator in a service-oriented system. The former is applied when a service needs to be replaced. The latter is applied to change the coordination sequencing of the coordinator and could therefore also involve adding a new service(s).

The transaction-based self-adaptation pattern could also be extended to multiple-phase commit protocols such as a three-phase commit protocol. A related pattern is that of a stateful interaction with an individual service consisting of more than one access to the service. This can be handled as a session between a client and a service. In this case, the adaptation connector for a stateful service must monitor the message to initiate a new session and the final response to end the session. Service adaptation can only be carried out when a session has been completed. New service requests are queued until the service adaptation is complete and the new service is operational.

Other adaptation patterns are the distributed coordination and hierarchical coordination adaptation patterns. Distributed coordination of the SOA application consists of multiple coordinators that are distributed and need to cooperate with each other. An example is an Emergency Response SOA where a city emergency coordinator seeks help from a neighboring city emergency coordinator. This coordination pattern is similar in behavior to the decentralized control pattern in distributed component-based systems, which was described previously in [6]. A coordinator in this adaptation pattern notifies its neighboring coordinator components when it needs to be adapted. The neighboring components cease to communicate with this component but can continue with other processing. Once the component has been replaced, its successor can resume communication with its neighbors.

A related research area is dynamic adaptation of service-oriented product lines, in which the different software configurations are organized as a product line, with dynamic adaptation from one member configuration to another managed through a feature model [26].


~/SASSY/work/runtime/demo/TwoPhaseCommitDemo
Requests were forwarded
Transitioned to "Quiescent" state after responsed
Request was enqueued in "Quiescent" state
Request in Queue was forwarded when transitioned to "Processing" state

```
Server ready...
Changing state from IdleState to ProcessingState
A new transaction is started. t = 1
Changing state from ProcessingState to PassiveState
A transaction is ended. t = 0
Changing state from PassiveState to QuiescentState
Jan 14, 2012 3:45:11 PM org.mortbay.log.Slf4jLog info
INFO: seeing JVM BUG(s) - cenceling interestOps==0
Service request is enqueued. q = 1
Changing state from QuiescentState to ProcessingState
Reactivated. New endpoint = http://localhost:9297/TwoPhas
ce/
A new transaction is started. t = 1
A transaction is ended. t = 0
Changing state from ProcessingState to IdleState
```

Figure 8: Execution trace for Coordinator Connector in two phase commit pattern

IX. Conclusions

This paper has described how software adaptation can be applied to distributed transactions in service oriented systems to dynamically adapt services and coordinators at run-time. This distributed transaction pattern addresses situations in which services and coordinators are stateful and follows the Two-phase Commit Protocol. The adaptation patterns are described by adaptation state machines encapsulated in coordinator and service adaptation connectors. In addition to service-oriented systems, this self-adaptation pattern could be applied to other types of distributed systems that participate in distributed transactions.

The main contributions of this paper are:

1. SOA adaptation patterns: in particular this paper has described the self-adaptation pattern for distributed stateful transactions in service-oriented systems and uses the two-phase commit protocol.
2. Design of adaptation connectors: adaptation connectors encapsulate the adaptation state machines for the two-phase commit adaptation pattern to separate the concerns of an individual service from software adaptation. As a result, the SOA adaptation connectors described in this paper are reusable regardless of the implementation of coordinators and services.

Future work will consist of investigating performance issues of dynamic adaptation for service-oriented architectures, developing additional adaptation patterns, and considering the failure of stateful services. Future research also includes automating the change management layer to automatically generate a sequence of adaptation commands by extracting the difference between the new software architecture and the original one.

Acknowledgment

This research is partially supported by grant CCF-0820060 from the National Science Foundation. The authors gratefully acknowledge the invaluable contributions of D. Menasce, S. Malek, J. Sousa, N. Esfahani, and J. Ewing to the SASSY project.

References

[1] H. Gomaa, “Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures”, Cambridge University Press, 2011.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, “Pattern Oriented Software Architecture: A System of Patterns”, John Wiley & Sons, 1996.

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison Wesley, 1995.

[4] D. Garlan and B. Schmerl, “Model-based Adaptation for Self-Healing Systems”, Proc. Workshop on Self-Healing Systems, ACM Press, Charleston, SC, 2002.

[5] H. Gomaa and M. Hussein, “Software Reconfiguration Patterns for Dynamic Evolution of Software Architectures”, Proc. WICSA Conference, Oslo, Norway, June, 2004.

[6] H. Gomaa and M. Hussein, “Model-Based Software Design and Adaptation”, Proc. ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, Minneapolis, MN, May 2007.

[7] H. Gomaa, “Designing Software Product Lines with UML:From Use Cases to Pattern-based Software Architectures”, Addison-Wesley, 2005.

[8] H. Gomaa, “A Software Modeling Odyssey: Designing Evolutionary Architecture-centric Real-Time Systems and Product Lines”, Keynote paper, Proc. 9th Intl. Conf. on Model-Driven Engineering, Languages, and Systems (MoDELS), Genova, Italy, Oct. 2006.

[9] J. Kramer and J. Magee, “The Evolving Philosophers Problem: Dynamic Change Management”, IEEE Transactions on Software Eng., Vol. 16, No. 11, 1990.

[10] J. Kramer and J. Magee, “Self-Managed Systems: an Architectural Challenge”, Proc Intl. Conference on Software Engineering, Minneapolis, MN, May 2007.

[11] M. Kim, J. Jeong, and S. Park, “From Product Lines to Self-Managed Systems: An Architecture-Based Runtime Reconfiguration Framework,” Proc. Design and Evolution of Autonomic Application Software, ICSE05, St. Louis, MO, May 2005, pp. 66-72.

[12] J. Lee and K. Kang, “A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering,” Proc. 10th Int. Soft. Product Line Conf. (SPLC 2006), Baltimore, Maryland, 2006.

[13] A. J. Ramirez and B. H. Cheng, “Applying Adaptation Design Patterns,” Prof. 6th Intl. Conf. on Autonomic Computing (ICAC), pp. 69-70, Jun. 2009.

[14] G. Li, et al., “Facilitating Dynamic Service Compositions by Adaptable Service Connectors”, International Journal of Web Services Research, Vol. 3, No. 1, 2006, pp. 67-83.

[15] F. Irmert, T. Fischer, K. Meyer-Wegener, “Runtime adaptation in a service-oriented component model”, Proc. Intl. Workshop on Software Engineering for Adaptive and Self-Managing Systems, May 2008, pp. 97-104.

[16] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, “Service-Oriented Computing: State of the Art and Research Challenges”, Computer, vol. 40, pp. 39-45, 2007.

[17] E. Thomas. Service-Oriented Architecture. Prentice Hall PTR, Upper Saddle River, 2005.

[18] E. Gat, Three-layer Architectures, “Artificial Intelligence and Mobile Robots”, MIT/AAAI Press, 1997.

[19] M. Kim et al., “Service Robot Software Development with the COMET/UML Method”, IEEE Robotics and Automation, Vol. 16, No. 1, March 2009, pp. 34-45.

[20] S. Malek, N. Esfahani, D. Menascé, J. Sousa, and H. Gomaa, “Self-Architecting Software Systems (SASSY) from QoS-Annotated Activity Models”, in Proc ICSE PESOS Workshop, Vancouver, Canada, May 2009.

[21] N. Esfahani, S. Malek, J. Sousa, H. Gomaa, and D. Menascé, “A Modeling Language for Activity-Oriented Composition of Service-Oriented Software Systems”, Proc. ACM/IEEE MODELS Conference, Denver, Colorado, Oct. 2009.

[22] D. Menascé, J. Ewing, H. Gomaa, S. Malek, and J.Sousa, “A Framework for Utility-Based Service Oriented Design in SASSY”, Proc. First Joint WOSP/SIPEW International Conf. on Performance Engineering, Jan. 2010.

[23] H. Gomaa, K. Hashimoto, M. Kim, S. Malek, D. Menascé, "Software Adaptation Patterns for Service-Oriented Architectures", Proceedings of ACM Symposium on Applied Computing (SAC), March 2010, Sierre, Switzerland.

[24] D. Menascé, J. P. Sousa, S. Malek, and H. Gomaa, “QoS Architectural Patterns for Self-Architecting Software Systems”, 7th IEEE Intl. Conf. on Autonomic Computing and Communication, Washington, DC, June, 2010.

[25] D. Menasce, H. Gomaa, S. Malek, J. Sousa, SASSY: A Framework for Self-Architecting Service-Oriented Systems", IEEE Software, Vol. 28, No. 6, November/December 2011.

[26] H. Gomaa and K. Hashimoto, “Dynamic Software Adaptation for Service-Oriented Product Lines”, in Proc. International Workshop on Dynamic Software Product Lines, Munich, Germany, August 2011.

[27] P. Bernstein, “Principles of Transaction Processing”, Second Edition, Morgan Kaufmann, 2009.