# BUST: Enabling Scalable Service Orchestration

Dong Liu, Ralph Deters
Department of Computer Science, University of Saskatchewan
Saskatoon, Saskatchewan, S7N 5C9 CANADA
dong.liu@usask.ca, deters@cs.usask.ca

## ABSTRACT

Service-Orientation (SO) is a design and integration paradigm that is based on the notion of well defined, loosely coupled services. Within SO, services are viewed as computational elements that expose functionalities in a platform-independent manner and can be described, published, discovered, and consumed across language, platform, and organizational borders. SO principles emphasize composability, by which a set of services can be composed to achieve the desired functionality. Service orchestration is the dominate approach to service compositions. A key issue in implementing service orchestrations is their efficient concurrent execution.

This paper focuses on the scalability challenges of simultaneously executing many long-running service orchestration instances. We present a novel approach for implementing service orchestrations called BUST (Break-Up State Transition) that significantly improves processing rate and scalability.

## Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*client/server, distributed applications*; C.4 [**Computer Systems Organization**]: Performance of Systems—*design studies, performance attributes*

## General Terms

Design, Performance

## Keywords

SOA, services, service orchestration, BPEL, multithreading, performance, scalability, queueing model

## 1. INTRODUCTION

Service-Orientation (SO) is a design and integration paradigm that is based on the notion of well defined, loosely coupled services. Within SO, services are viewed as computational elements that expose functionality in a platform-independent manner and can be described, published, discovered, orchestrated and consumed across language, platform and organizational borders [20, 4]. The Service-Oriented Architecture (SOA), first introduced by Gartner in 1996, is a conceptual framework that identifies service consumers, service providers, and service registries. Web Services (WS) are the most commonly used technologies to develop SO applications, due to the standardization efforts and the available tools and infrastructures.

SO principles emphasize service composability, by which a set of services can be composed to achieve the desired functionality. The service compositions are also called service assemblies [8]. A service participating in a composition is called a composition member. It is important to note that service orchestration and service choreography are two fundamentally different approaches for composition. Service orchestration depends on a conductor-like central service that executes a plan/workflow that defines when and how the members are to act. Service choreography, on the other side, does not have a central service and is based on the idea that the members are able to interact according to a previously agreed choreography description. Consequently choreography assumes that the members are collaborative enough to be able to achieve the common goal while the members of a service orchestration do not need to know any details about the orchestration and is therefore much easier to implement. Not surprisingly, service orchestration is the dominant approach for achieving service compositions. To explain service orchestration in more detail, Figure 1 shows the messaging sequence of a simple service orchestration for loan application. The node with an 'O' inside is the service orchestration. The numbers within the message names denote different messages and their sequence. A message may have a corresponding response that is denoted by an 'r' after the sequence number. Two messages with the same first sequence number are sent in parallel, e.g. 2.1 and 2.2.

As can be seen, a client sends a loan application with the personal information to the loan approval service (LAS), and the LAS checks the client's credit by invoking the credit checking service. If the credit rating is good, the LAS sends the application to two loan suppliers, and the loan offer with lower interest will be sent back to the client. Typically this business process will be represented in orchestration language like the Web Services Business Process Execution Language (WSBPEL, formerly known as BPEL4WS, BPEL for short) [21] to specify all details for the automated service


INFOSCALE 2007, June 6-8, Suzhou, China

DOI 10.4108/infoscale.2007.901

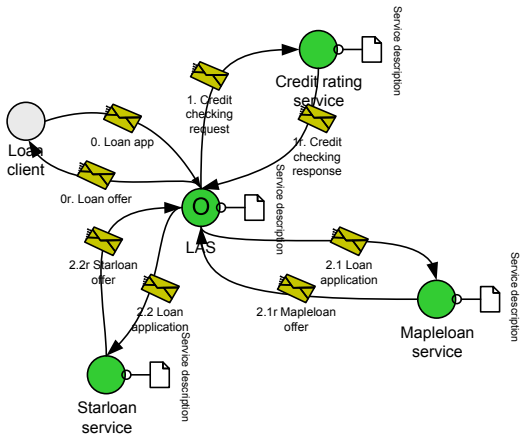


**Figure 1: The loan application service orchestration.**

orchestration. To execute the BPEL-represented business process a compiler can convert the orchestration into code for execution on a service platform. The deployed orchestration and the hosting service platform together determine the performance of the orchestration executions.

Current service platforms favor the use of multithreading as a means for dealing with simultaneous requests. This improves the performance of the platform compared to single-threaded implementations [23]. The service discipline of the multithreading service platform is processor sharing (PS). Figure 2 shows the throughput and average residence time of a web service as the functions of the number of concurrent active threads. The web service is implemented in Axis2[1] and runs on Jetty 5.0[2]. The number of concurrent active threads for service processing is equal to the number of concurrent requests in the experiments because Jetty assigns a thread for each request. The residence time does not include the waiting time spent in queues, because the queue length has been set to be zero in the experiments. As shown, the average request residence time increases linearly with the number of concurrent active threads in the system before the system thrashes. Similar results have been observed on .NET applications [12] and Java EE applications [24, 11]. The number of concurrent active threads will increase if it takes a long time for some threads to finish their jobs and the job arrival rate does not change. The performance of a service platform degrades when the number of concurrent active threads increases. This is a serious scalability problem for service orchestrations because the threads for orchestration instances need to run longer if a service orchestration comprises more message exchanges with the members. An orchestration instance refers an execution of the service orchestration initialized by a specific service request.

Generally, a thread has two states, active and idle, after it is spawned and before it is terminated. When a thread is active, it will affect the performance of other threads. On the contrary, an idle thread has very little impact on the performance of active threads. When a thread switches from the active state to the idle state, the job information that it carries is removed. The top part of Figure 3 shows the sequence of messaging events and state transitions of a thread for an instance of the common LAS orchestration implementation. The thread responsible for a service instance is active from

[1]See `http://ws.apache.org/axis2/` .
[2]See `http://jetty.mortbay.org/index.html` .

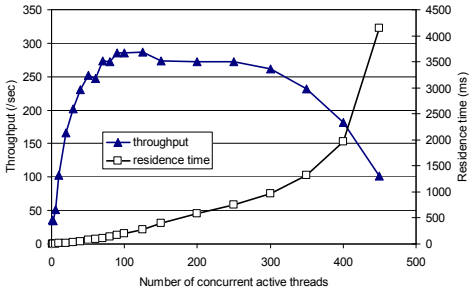


**Figure 2: The throughput and residence time of a web service versus the number of concurrent active threads.**

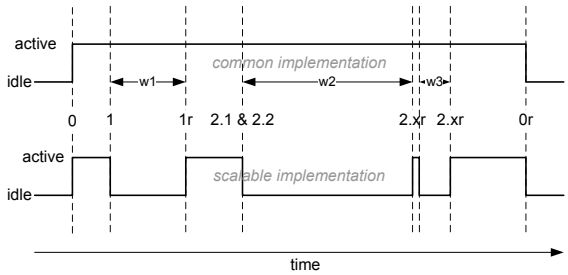


**Figure 3: The sequence of messaging events and state transition of a thread for common and scalable implementation of the LAS orchestration. The first 2.xr can be either 2.1r or 2.2r, and the second will be the other.**

receiving the '0. Loan app' message till sending the '0r. Loan offer' message. During the periods of w1, w2, and w3, the thread is waiting for a message. If the thread can be set to the state of idle during w1, w2, and w3 as shown in the bottom part of Figure 3, the number of concurrent active threads will be decreased, and the service platform performance will be improved. However, there are three major problems for achieving a scalable implementation:

1) How to represent a service orchestration in such a way that the possible idle periods between the active periods can be identified;

2) How to ensure that an inbound message triggers the right idle-to-active state transition of an orchestration instance;

3) How to maintain the state of an orchestration instance through the different processing periods.

This paper proposes a novel Break-Up State Transition (BUST) approach to enable scalable service orchestration implementations. The principles of BUST are presented in Section 2. Section 3 discusses how to apply BUST to BPEL orchestrations with a case study. Section 4 evaluates BUST by simulation. The related work is reviewed in Section 5. Section 6 is the conclusion and future work.

## 2. BUST: BREAK-UP STATE TRANSITION

In the following a description of the BUST approach for dealing with the inherent scalability problems is given.

### 2.1 Break-Up

Service orchestration can be considered as the flow of message exchanges between the central conductor service and the member services. Consequently, service orchestration is composed of processing of inbound messages and generation

of outbound messages. Normally, the processing happens consecutively with a messaging or timing event. Figure 4 shows a generic model for the interactions between an elementary service unit and its environment. The model is developed on the basis of an interaction model used in software requirements analysis [7]. The service component first validates a messaging or timing event. If the event is validated, corresponding processing will be triggered, and a generated outbound message may be sent. In implementation, the event drives a thread to switch from the idle state to the active state. The thread turns idle when the processing is finished or a message is sent.

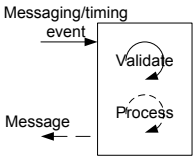


**Figure 4: The model for an interaction between an elementary service component and its environment.**

The model shown in Figure 4 is generic enough to describe any interaction scenarios for a service orchestration. Therefore, it is the template for the snippets into which a service orchestration can be disassembled. One can verify the model's capability by using it to describe the basic message exchange patterns (MEP) of service interactions. There are four basic MEP's, in-only, out-only, in-out, and out-in [26]. The in-only and out-only patterns are for one-way messaging. The in-out pattern is also referred as request-response, and the out-in pattern is referred as solicit-reply. Figure 5 shows the MEP's described by the interaction template. It is straightforward to describe the in-only and in-out patterns using the template. For out-only pattern, the outbound messaging is combined with an inbound messaging or timing event that triggers the generation of the outbound message. The out-in pattern is described by linking the out-only pattern and the in-only pattern together.

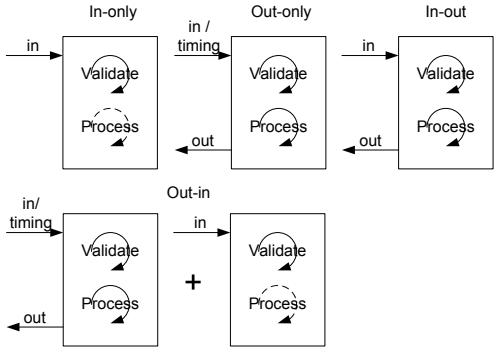


**Figure 5: Four MEPs described by the interaction model.**

A general rule governing the application of the interaction template for disassembling a service orchestration is to ensure that there is no idle period inside a snippet. An idle period is always introduced by waiting for an inbound message or a timing event. The snippets should be implemented separately, and each snippet has its own service endpoint (called a snippet endpoint). There is no centralized control for the execution sequence of snippets. The snippets can be deployed on either the same service platform or several service platforms. Figure 6 shows the break-up result of the

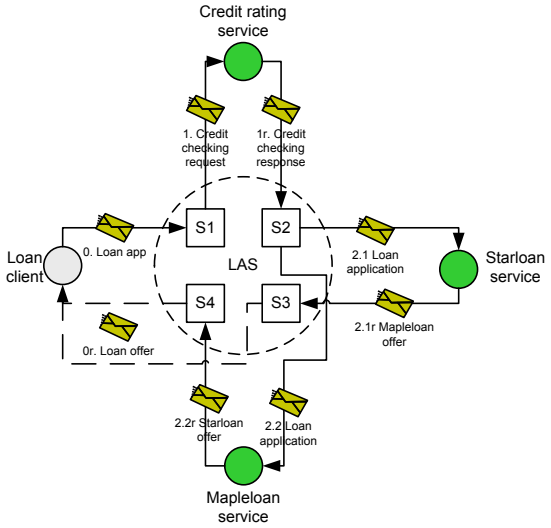


**Figure 6: The disassembled LAS orchestration. S1, S2, S3, and S4 are four snippets. Note that either S3 or S4 sends the message '0r. Loan offer' to the loan client.**

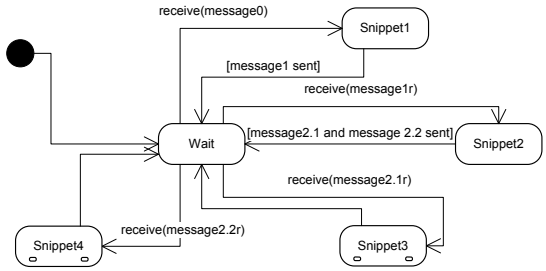


**Figure 7: The state machine diagram of LAS orchestration.**

example LAS orchestration. Section 3 will present more detailed rules for breaking up a service orchestration in the BPEL language.

## 2.2 State Transition

When a service orchestration is disassembled into snippets, its execution transforms into the execution the snippets with idle periods in between. This execution can be modeled by a virtual state machine. Each snippet is corresponding to a state in the state machine. There is also a special state called "Wait" representing that the machine halts and wait for a messaging or timing event to happen. Figure 7 shows the diagram for the state machine of LAS orchestration using UML notations [3].

Since there is no centralized control for the execution of the service orchestration, successful state transitions depend on the cooperation of the member services. A successful state transition has two requirements:

1) a message is delivered to the correct snippet endpoint, and
2) the message is correlated with the correct orchestration instance.

Each snippet puts the information about to which snippet endpoint a reply message should be sent in the messages that it sends to a member service. In this way, the first requirement is accomplished. Once an orchestration instance is initialized, the first executed snippet will create a

unique id for the instance. This id will be included in every message sent to the member services by each snippet. The member services are also required to include the id in the reply messages. So the correct orchestration instance can be identified when an inbound message is parsed. Each snippet endpoint should be able to correlate an inbound message to the corresponding orchestration instance and assign the reply-to endpoint address and instance id to the outbound messages.

## 2.3 State Management

Services are often referred to be stateless [8, 19]. The statelessness is a design principle of services to achieve scalability. In fact, no service is really stateless. The states have to be managed in many cases, especially for service orchestrations. State management is critical for the following scenarios in service orchestration executions:

1) the service orchestration correlates a message to an instance based on the identifiers of messages and instances;
2) the execution logic of a service orchestration depends on the runtime information included in messages; and
3) the details of interactions with business partners need to be stored for later access.

For BUST-style service orchestration implementations, the states can be classified into two categories by their lifetime. Some states exist with a message. When the message is received and processed, the states disappear. Other states persist across the execution of several snippets or the whole orchestration. These states are critical for the transitions of the virtual state machine and the inner logic of snippets. The first type of states does not need to be specially managed as far as the message transportation is reliable. We need to consider the management of the second type of states. Without the loss of service scalability, there are two methods to maintain the states:

1) the messages carry the state information across the snippets; and
2) the state information is maintained by a persistent storage, like a database, with which the snippets can interact.

BUST depends on a persistent storage to store the states that can be accessed by all snippet endpoints. BUST favors representing the orchestration states and storing the entries in the persistent storage explicitly. Object persistence is not good for state persistence in BUST because it will increase the coupling between snippets and degrade scalability. Some states, e.g. the reply-to snippet endpoint address and the instance id, should always be carried by the messages, because the messages are also means to trigger the transitions of the virtual state machine. The case study in Section 3 shows how to apply these principles to a BPEL orchestration implementation.

# 3. APPLYING BUST TO BPEL IMPLEMENTATIONS

BPEL is a widely used description language for web service orchestrations. The members of a BPEL orchestration are often called partners in business context. A BPEL orchestration is composed of scopes and activities. A scope provides context for the activities inside. Table 1 lists major basic activities and structure activities defined in BPEL version 2.0. The orchestration's messaging and timing events

```
receive(client, loanApp)
assign(loanApp/SIN, creditRatingRequest/SIN)
invoke(creditRatingService, creditRatingRequest,
            creditRatingResponse)
assign(creditRatingResponse/creditRating,
            loanApplication//creditRating)
flow{
   sequence{
      invoke(mapleLoanService, loanApplication)
      receive(mapleLoanService, mapleLoanOffer)
   }
   sequence{
      invoke(starLoanService, loanApplication)
      receive(starLoanService, starLoanOffer)
   }
}
if(mapleLoanOffer/interest<=starLoanOffer/interest){
   assign(mapleLoanOffer, loanOffer)
else
   assign(starLoanOffer, loanOffer)
}
reply(client, loanOffer)
```

**Figure 8: The BPEL description of the loan approval service orchestration in pseudocode. / and // are notations in XPath. / is the child operator. A/B denotes B that is an immediate child of A. // is the recursive descent operator. A//B denotes B that is the first children found in the zero or more levels of B.**

**Table 2: BPEL Basic Activities Break-Up Rules**

| Activity | Break-up results | |
|---|---|---|
| ...<br>invoke(S, mOut, mIn)<br>... | ...<br>invoke(S, mOut)<br>//end | //start<br>receive(S, mIn)<br>... |
| ...<br>receive(S, mIn)<br>... | ...<br>//end | //start<br>receive(S, mIn)<br>... |
| ...<br>waitFor(ms)<br>... | ...<br>register(Timer, SYSTEM.current + ms) //end | //start<br>receive(Timer, alarm)<br>... |
| ...<br>waitUntil(time)<br>... | ...<br>register(Timer, time)<br>//end | //start<br>receive(Timer, alarm)<br>... |

No break-up operations are needed for reply(), assign(), and exit(). ···'s are the context of an activity. Timer is an internal service that accepts registrations of timing events and sends alarms. SYSTEM.current is the systems current time. register(Timer, time) registers a timing event specified by time on the Timer.

are represented by the basic activities. Figure 8 shows the BPEL description of the LAS orchestration example. The sections of partner links and variables are not included in the description.

## 3.1 BPEL Break-up

The general principle for disassembling a BPEL orchestration is that a snippet always starts with an inbound messaging or timing event. Table 2 lists the break-up rules for the basic activities.

## 3.2 State Transition by WS-Addressing

As discussed in Section 2.2, the state transitions involve message deliveries and instance correlations in the virtual machine of snippets. BUST solves the two problems together by applying the Web Service Addressing (WSA) to the inbound and outbound messages of BPEL orchestrations. The WSA specification describes a transport-neutral mechanism for the addressing issues of web services and mes-

**Table 1: BPEL Activities and Corresponding Pseudocode**

| Type | BPEL notation | Meaning | Pseudocode |
|---|---|---|---|
| Basic activities | <invoke> | Invoke an operation of a partner service. The MEP can be either in-out or in-only. | invoke(S, mOut, mIn) or invoke(S, mOut) |
| | <receive> | Receive a message from a partner. | receive(S, mIn) |
| | <reply> | Reply a message to the client or a partner service. | reply(S, mOut) |
| | <assign> | Update a variable. | assign(xA, xB) // xB:=xA |
| | <wait> | Delay for a period or until a deadline. | waitFor(ms), or waitUntil(time) |
| | <exit> | End the execution of the instance. | exit() |
| Structur activities | <sequence> | The inside activities are executed sequentially. | sequence{} |
| | <if> | Condition structure. It can be combined with <elseif> and <else>. | if(condition) {} else {} |
| | <while> | Repeat the execution of inside activities. | while(condition) |
| | <pick> | Execute an activity on a messaging or timing event in a set of them. | pick{onMessage(S, mIn) {}} |
| | <flow> | Concurrency and synchronization. | flow{} |

S: service partner, mIn: inbound message, mOut: outbound message, ms: time duration in millisecond, time: a time point, {}: a segment containing one or more activities.

sages [25]. A SOAP message header contains elements like <To>, <From>, <ReplyTo>, <MessageID>, and <RelatesTo> when the WSA is engaged. We take advantage of the elements of <ReplyTo>, <MessageID>, and <RelatesTo> for state transitions in BUST.

A SOAP message sent out by a BPEL orchestration must have a <ReplyTo> header element. The <ReplyTo> element contains the endpoint address of the snippet that is supposed to receive the corresponding response message. In this way, when the service partner receives the message, it knows where the response message should be sent to. This requires to implement service endpoints for each of the receive(S, mIn) activity in snippets, including the one resulted from the disassembling of invoke(S, mIn, mOut).

A SOAP message sent out by a BPEL orchestration must have a <MessageID> header element. According to the WSA specification, the content of <MessageID> in the request message should be copied to the <RelatesTo> element in the corresponding response message. In BUST, the <MessageID> contains a string composed of a unique identifier for the message and a unique identifier for the orchestration instance, like <MessageID>messageid:instanceid </MessageID>. A partner service will have no awareness of what the identifier string means in the request message, and it just put the id string in the <RelatesTo> elements of the response messages to the snippet endpoint specified in <ReplyTo>. The snippet can parse out the instanceid from an inbound message and correlates it to the correct orchestration instance.

The transitions triggered by timing events are achieved similarly like messaging. The Timer works as an internal service. When a snippet registers a timing event on the Timer, the information about the snippet endpoint to receive the timing alarm and the orchestration instance id are included in the registration request. Therefore, the Timer will know to which snippet and instance it sends the alarm.

## 3.3 Explicit State Management

The state information of a BPEL orchestration can be classified into two types, the states represented by <variable> elements and those that are not explicitly described by BPEL. In BUST, all the states that need to be managed are explicitly represented. The BPEL syntax is extended by adding features of state management activities. The extension does not change the original semantics of BPEL, and therefore the BPEL compilers and code generators can be reused for BUST-style implementations. Table 3 lists the extension activities for state management.

**Table 3: BPEL Extension for State Management**

| Notation | Meaning | Pseudocode |
|---|---|---|
| <create> | Initialize an instance by specify the database and the instance id. An entry for the instance will be created in the database. | create(DB, instanceID) |
| <update> | Update certain state of an instance in the database. | update(DB, instanceID, ID, state) |
| | Update the value of the variable. | ID: variableName<br>state: value |
| | The message has been sent. Or a message related to the id has been received. | ID: messageID<br>state: transportation |
| | For a <flow> structure, the state is the number of concurrent activities that have finished. | ID: structureID<br>state: syncNum |
| | For a <while> structure, the state is whether the current message will finish the loop. | ID: loopID<br>state: finished |
| | For a <pick> structure, the state is whether the current message will triggered a picked activity. | ID: pickID<br>state: picked |
| <retrieve> | Check certain state of an instance in the database. There are similar cases for <retrieve> to those of <update>. | retrieve(DB, instanceID, ID, state) |

The final break-up result of the LAS orchestration are shown in Figure 9. Although Snippet4 is very similar to Snippet3, they cannot be merged together because the interfaces to receive messages from the Starloan service and the Mapleloan service can be different.

# 4. EVALUATION

Most current service delivery systems are implemented on the basis of a three-tier client-server architecture [16]. The architecture of such service delivery systems is shown in Figure 10. A model will probably miss the key characteristics of service performance if it includes all the components in Figure 10. It is better to just focus on the service platform. Most service platforms work in a multithreading fashion. The threads are controlled by the application server on which a service platform is deployed. A number of threads are created and put in a thread pool when an application server is initialized. When an arrival request is dispatched to a thread in the pool, the thread becomes active. An active thread will return to the idle state when its job is finished. When the number of idle threads in the pool is less than a minimum number, new threads will be spawned and put into the pool. However, the total number of threads including both active and idle ones in the pool can never be more

than a maximum number bound. When too many threads are idle, some idle threads will be destructed and removed from the pool if no new arrivals are dispatched in time. The application server can prevent too many threads from using up the resource by such a bounded thread pool. The state transition diagram of a thread is shown in Figure 11. A thread consumes the computing resource when it is active.

## 4.1 A Queueing Model for Common Web Service Platforms

The Queueing Network Model (QNM) has been widely used for software system performance evaluation [22, 2]. The advantage of applying the QNM to software performance evaluation is that the performance of a large-scale complex system can be precisely predicted based on the knowledge of its architecture and performance characteristics of its components. Hence, the service orchestration is an appropriate case to apply the QNM. First of all, it is necessary to model the service platforms. Figure 12 shows the architecture of typical web service platforms using HTTP as the transportation methods.

We use AnyLogic 5.5[3] to develop the QNM models and run the simulation experiments. Figure 13 shows a QNM for service platforms. The TCP layers and the HTTP sender are not included in the model because they have little impact on service performance in most cases. The HTTP listener is a queue to accept requests. The dispatcher assigns a request to an idle thread in the thread pool. Then the request is processed. When the processing is finished, the job departs the system and the thread will return to the idle state. In experiments with Jetty, we found that the threads are spawned so fast that there will always be idle threads in the thread pool when a request waits for being dispatched before the maximum thread number is reached. There is very small overhead for thread spawn and destruction when the arrival rate is relatively stable. Since the thread pool has less influence on the performance of the service platform, the model excludes the thread pool without loss of precision.

**Table 4: Service Platform QNM Configurations**

| Component | Configuration |
|---|---|
| Arrival | In an open network, we assume the inter-arrival time is exponential distributed, i.e. Poisson arrivals. The parameter is the average arrival time $\lambda_a$. |
| HTTP listener | The listener is a limited capacity queue. The service discipline is FIFO (First In First Out). The arrivals in the queue will expire if they cannot be served in time. When the queue is full, new arrivals are rejected. The capacity is $C_l$. The timeout duration is $TO$. |
| Dispatcher | When there is an idle thread, the dispatcher assigns the first arrival in the listener queue to the thread. This happens very fast. Therefore, we do not count the time. |
| Processing | We assume the service discipline is PS, and the service time is Erlang distributed. The parameters are the rate $\lambda_p$ and the shape $k$. The capacity of the processing component, i.e. the thread pool maximum number bound, is $C_p$. |
| Departure | Once its processing finished, the request leaves the system immediately. |

The details about the model configurations are listed in Table 4. It would be the most straightforward to assume that the service time distribution is exponential distributed, which yields an M/M/1 queue. However, we found that the service time distribution of the PS service component is

[3]See `http://www.xjtek.com/anylogic/` .

```
//Snippet1
receive(client, loanApp)
create(DB, instanceID)
update(DB, instanceID, loanAPP)
assign(loanApp/SIN, creditRatingRequest/SIN)
update(DB, instanceID, creditRatingRequest)
invoke(creditRatingService, creditRatingRequest)
update(DB, instanceID, creditRatingRequest//MessageID)

//Snippet2
receive(creditRatingService, creditRatingResponse)
update(DB, instanceID, creditRatingResponse//RelatesTo)
update(DB, instanceID, creditRatingResponse)
assign(creditRatingResponse/creditRating,
            loanApplication//creditRating)
update(DB, instanceID, creditRatingResponse)
flow{
  sequence{
    invoke(mapleLoanService, loanApplication)
    update(DB, instanceID, loanApplication//MessageID)
  }
  sequence{
    invoke(starLoanService, loanApplication)
    update(DB, instanceID, loanApplication//MessageID)
  }
}

//Snippet3
receive(mapleLoanService, mapleLoanOffer)
update(DB, instanceID, mapleLoanOffer//RelatesTo)
update(DB, instanceID, mapleLoanOffer)
update(DB, instanceID, flowID, syncNum)
retrieve(DB, instanceID, flowID, syncNum)
if (syncNum == readyNum) //in this case readyNum is 2
{
  retrieve(DB, instanceID, starLoanOffer)
  if(mapleLoanOffer/interest<=starLoanOffer/interest){
    assign(mapleLoanOffer, loanOffer)
    update(DB, instanceID, loanOffer)
  else
    assign(starLoanOffer, loanOffer)
    update(DB, instanceID, loanOffer)
  }
  reply(client, loanOffer)
  update(DB, instanceID, loanOffer//MessageID)
}

//Snippet4
receive(starLoanService, starLoanOffer)
update(DB, instanceID, starLoanOffer//RelatesTo)
update(DB, instanceID, starLoanOffer)
update(DB, instanceID, flowID, syncNum)
.../the rest is similar to Snippet3
```

**Figure 9: The snippets got from the application of BUST to the BPEL description of the LAS orchestration.**

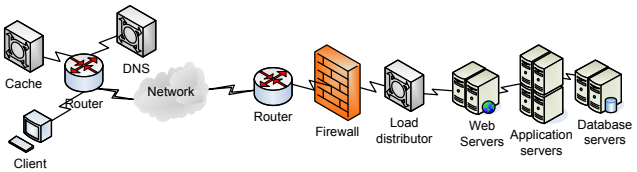


**Figure 10: The architecture of service delivery systems.**

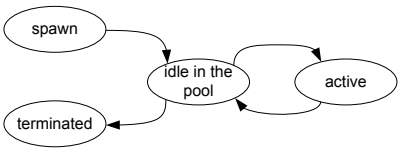


**Figure 11: The state transition diagram of a thread.**

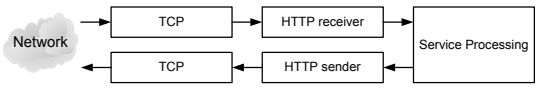


**Figure 12: The architecture of a typical web service platform.**

different from an exponential distribution in the experiments with Jetty and Axis2. It has a heavier tail than exponential. Generally, the PS queue can be modeled as M/G/1 [6]. The Pareto distribution and the Gamma distribution are used as the service time distribution is some approaches [13, 5]. We use the Erlang distribution for the service time and model the system as an $M/E_k/1$ queue. The probability density function (pdf) of the Erlang distribution is

$$f(x) = \frac{\lambda^k x^{k-1} e^{-\lambda x}}{(k-1)!},\ x \geq 0,\ \lambda > 0,\ k \in \mathbb{N},$$

where $\lambda$ is the rate parameter, and $k$ is the shape parameter. The mean of the random variable $x$ is

$$\mathrm{E}[X] = k/\lambda\,.$$

Figure 14 shows the cumulative distribution functions (CDF) of the experiment data fitted by the Erlang distribution compared with the exponential distribution. The empirical data was the measurement result of the service time of a web service implemented in Axis2 and runs on Jetty 5.0. The workload was generated by Apache JMeter[4] when 25 concurrent clients were simulated. Obviously, the Erlang distribution is a much better fit than the exponential distribution.

In the model, we let $k$ equal to the number of active threads, and estimate $\lambda$ by

$$\lambda = k/\bar{S}_k\,,$$

where $\bar{S}_k$ is the average service time of $k$ concurrent active threads obtained from experimental measurement. We found that $\bar{S}_k$ is linearly proportional to the number of active threads $k$ when the system is running in normal operational state. Therefore, we can approximate $\lambda$ by

$$\lambda = 1/\bar{S}_1\,,$$

where $\bar{S}_1$ is the average service time when there is only one active thread in the system.

## 4.2 Queueing Models for the LAS Orchestration

Figure 15 shows a QNM developed in AnyLogic for the common implementation of the LAS orchestration. We did not explicitly include the network latencies in the model because their impacts are trivial compared with the processing delays at the partner services and they can be combined with

[4]See `http://jakarta.apache.org/jmeter/index.html` .

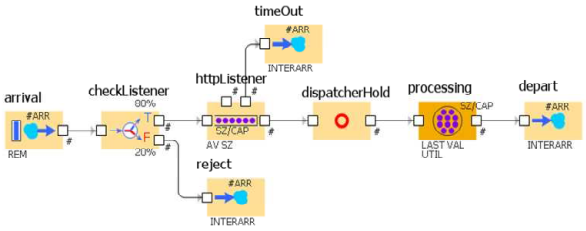


**Figure 13: A simplified QNM for service platforms.**

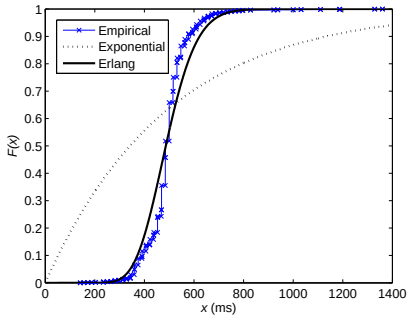


**Figure 14: Fitting the service time of a web service using the exponential distribution and the Erlang distribution.**

the partner services delays in needed. The 'router' transports the messages to the proper services. The 'flow' component makes a copy for each message pass it, and sends them to the starLoan and mapleLoan component concurrently. A message needs to match with its copy at the 'synch' component, and then they are combined together. In this way, the <flow> structure is modeled, and the synchronization condition is assured. All the three partner services are of PS service discipline, and the service times follows the Erlang distribution. A message returns directly to the processing component without waiting for dispatching after it is processed by partner services because there is already a thread assigned for it. The number of active threads in the processing component increases by one for each new dispatched arrival, and decreases by one for each departure correspondingly.

Figure 16 shows a QNM for the LAS orchestration implemented in BUST style. The key difference between this model and the previous model is that the number of active threads decreases by one when a message leaves the processing component. The messages returning from the partner services need to go to the HTTP listener to wait for dispatching. The messages from the partner services have higher priorities than the new arrivals, and the new arrivals will be preempted if the listener queue is full.

We did not include a database component in the model because either CPU-intensive jobs or database-intensive jobs are processed in a processor sharing discipline, and the service time distributions for the two cases are very similar. Although the requirements of state management are explicitly represented in BUST, it does not mean that the BUST style implementation will need more processing time on the database, because state persistence is a common requirement for all service orchestration implementations [8].

## 4.3 Simulation Results and Discussion

In order to compare the BUST and common orchestration

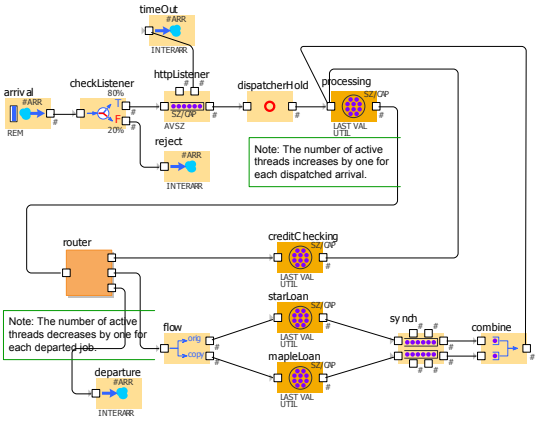


**Figure 15: A QNM for the common LAS orchestration implementation.**

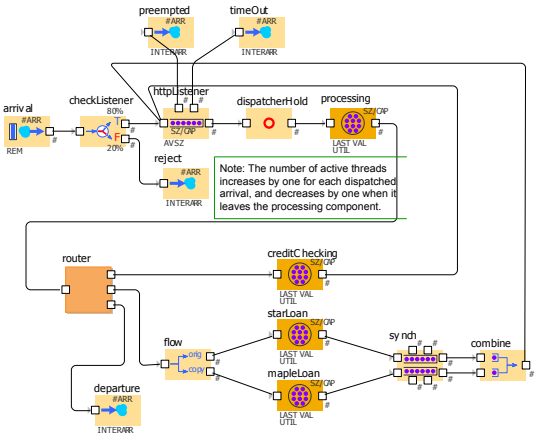


**Figure 16: A QNM for the BUST LAS orchestration implementation.**

implementation, we carried out a series of simulation experiments using the models developed in AnyLogic. Table 5 is a configuration of the model parameters. We assume the partner services are always busy and working at a stable state. Therefore their shape parameters of service time distributions are constant. In this configuration, the partner services all have relative fast processing rates compared to the orchestration.

Figure 17 shows the average residence time and the failure rate as a function of the arrival rate. The job residence time is the duration from its arrival into the HTTP listener queue till its departure. The failing jobs are the arrivals that are rejected at the HTTP listener, or are preempted from the HTTP listener queue, or leave the HTTP queue because of timeout. Each experiment simulates the system running for 20000 time units, which means about $2 \times 10^5$ arrivals at an arrival rate of 10 per time unit. We calculate the average residence time using a slide window of 5000 departures when the system is in stable operation state. The failure rate is the proportion of failed arrivals to all the arrivals. The simulation results indicate that the BUST orchestration implementation has shorter average residence time and lower failure rate than the common one throughout the tested range of the arrival rate. However, the difference between the two approaches is not significant.

Table 6 is the other configuration of the models. We only changed the parameters of partner services service time in

**Table 5: The First Configuration of the Models**

| Component | Configuration |
|---|---|
| Arrival | $\lambda_a$ ($\sec^{-1}$) changes to simulate different workload. |
| HTTP listener | $C_l = 100$, $TO = 30$ sec. |
| Processing | $\lambda_p = 50\,\sec^{-1}$, $k$ = the number of active threads, $C_p = 200$ . |
| Credit checking service | $\lambda = 1000\,\sec^{-1}$, $k = 100$, $C$ =infinity |
| Star loan service | $\lambda = 1000\,\sec^{-1}$, $k = 200$, $C$ = infinity |
| Maple loan service | $\lambda = 1000\,\sec^{-1}$, $k = 150$, $C$ = infinity |

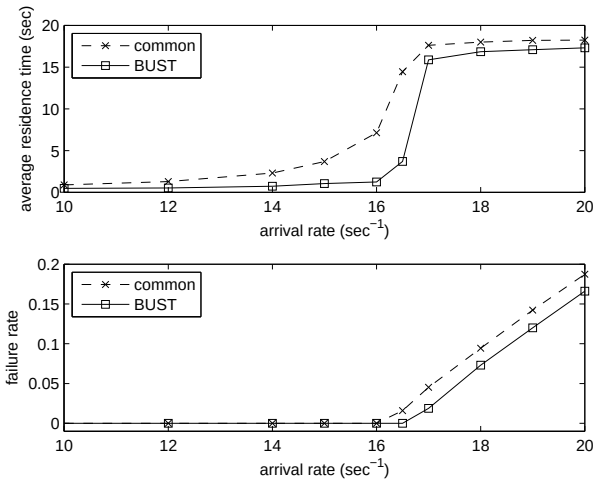


**Figure 17: The average residence time as a function of the arrival rate of common and BUST orchestration implementation. The models are configured as Table 5.**

such a way that some partners have relative slow processing rate compared to the orchestration.

Figure 18 shows the average residence time and the failure rate as functions of the arrival rate in the experiments using the configuration in Table 6. The figure indicates that BUST can improve the orchestrations performance dramatically when a jobs processing time spent at partner services is relatively long. This result perfectly accords with the design goals of BUST because the thread scalability problem gets worse when the threads need to wait longer for the messages from the partner services. BUST achieves the predominance by efficiently decreasing the number of concurrent threads in the service platform. More experiments are to be carried out in our future work to see how the system behaviors change with various partner service rates and arrival rates as well.

## 5. RELATED WORK

The Representational State Transfer (REST) architectural style, introduced by Fielding [10, 9], is an abstract model for the Web architecture and can be used to design loosely-coupled, extensible, and scalable web applications. Although many of current web services practices did not follow the REST principles, REST is still valuable to guide the design of web services applications. BUST is a design based on REST principles. In REST, a well-designed Web application can be viewed as a virtual state machine, where a user triggers the state transition by sending requests to the application through a web browser. In BUST, the service orches-

**Table 6: The Second Configuration of the Models**

| Component | Configuration |
|---|---|
| Credit checking service | $\lambda = 100\ \text{sec}^{-1}$, $k = 200$, $C$ =infinity |
| Star loan service | $\lambda = 25\ \text{sec}^{-1}$, $k = 200$, $C$ = infinity |
| Maple loan service | $\lambda = 20\ \text{sec}^{-1}$, $k = 200$, $C$ = infinity |

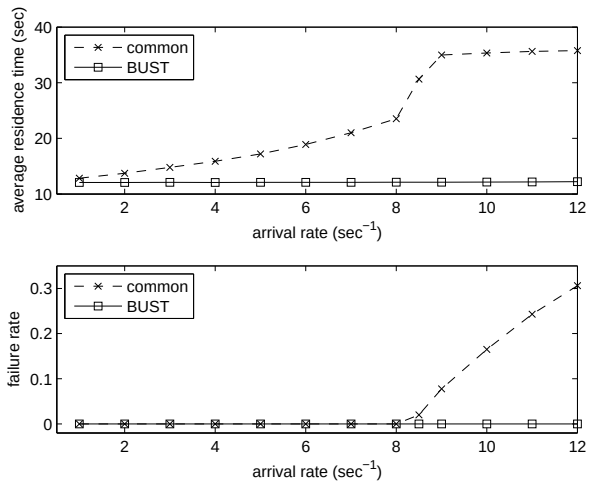


**Figure 18: The average residence time as a function of the arrival rate of common and BUST orchestration implementation. The models are configured as Table 6.**

tration is a similar virtual state machine, where a partner service triggers the state transition by sending messages to it.

Spring Web Flow (SWF) is a component of Spring Framework's web stack[5]. Like a service orchestration, a web flow involves a number of HTTP requests, is stateful, and is often dynamic and long-running. In SWF, the flow execution key and the event identifier are used to correlate the request to a started flow execution and resume it. BUST applies a similar method for instance correlation. SWF has an executor object that play the role of centralized controller for the flow execution. On the contrary, there is no centralized controller for the execution of a service orchestration in BUST.

A continuation is a representation of the state of a computation entity, i.e. a program or a process, at a point of its execution. A process can be stopped during execution and be resumed later with the help of a continuation. It is easy to control complex flows by using continuations. Some web applications use continuations to solve the scalability problem of threads by suspending a thread waiting for an event and resume it later[6]. Manolescu proposed an object-oriented (OO) workflow framework called micro-workflow by combining continuations and asynchrony together [15, 14]. The Jacob (Java Concurrent Objects) framework[7] provided a service orchestration execution virtual machine that was developed on the basis of the ACTORS model [1] and the Pi calculus [17]. Continuations and asynchrony are key features of the Jacob virtual machine. These approaches can be applied to tackle the thread scalability problem. However, it may bring other scalability problems. Continuations are implemented or simulated by objects in OO languages, and the objects need to contain messaging and orchestration execution contexts. That will create a huge memory footprint. And it gets worse when the continuation objects need to live for a long time. If the continuation objects are serializable, the memory footprint problem can be solved by saving the objects to and loading them from a persistent storage. However, this may cause compatibility problems when a service platform tries to resume the continuation object created and saved by the other service platform. It will be very difficult to execute an orchestration instance across several service platforms in a decentralized manner. BUST does not bring such problems, because the states are explicitly managed.

Welsh et al. proposed an architecture named SEDA (staged event-driven architecture) for scalable Internet services [28, 27]. Applications in the SEDA style are a network of event-driven stages connected by queues. A stage represents a robust building block with an event queue. SEDA aims to enable the Internet services to support massive concurrency and self-tuning resource management. The concept of stage in SEDA is similar to the concept of snippet in BUST. However, SEDA seems to focus on the macro architectural aspect of Internet services, while BUST focuses on service orchestrations and workflows. SEDA does not have the notion that the whole system can be viewed as a virtual state machine like those in REST and BUST.

Nanda et al. proposed an approach to partition a BPEL orchestration into decentralized processes [18]. The approach is based on a complex algorithm to partition program dependence graphs representing BPEL orchestrations. By this way, a BPEL orchestration can be deployed and executed in a decentralized fashion, which is supposed to decrease the load on each server that hosts part of the orchestration. However, this approach introduces a mass of message exchanges between the partitionings, which brings more loads to the partitionings. Decentralized and distributed deployment and executions of service orchestrations are naturally supported by BUST. Different from this approach, BUST avoids the cost of message exchanges between snippets by event-driven state transition. BUST also provides a much simpler mechanism to partition a BPEL orchestration into snippets.

# 6. CONCLUSIONS AND FUTURE WORK

This paper presents the BUST architectural style for scalable service orchestration implementations. BUST assimilates the essentials of other architectural styles like REST and SEDA, and applies them to service orchestrations. Our preliminary simulation experiments show that BUST can improve the system's performance dramatically when a service orchestration involves message exchanges with partner services of large latency. BUST is a promising approach to tackling the scalability problem of long-run complex service orchestrations.

We are now developing a BUST code generator that can disassemble a BPEL orchestration description into snippets in extended BPEL. We are also developing a BUST BPEL engine that can convert the snippets into Java packages deployable on Servlet containers. The mean value analysis (MVA) method will be applied to the performance evaluation of complex service orchestrations besides simulation in our future work.

[5]See `http://www.springframework.org/documentation` .
[6]See `http://docs.codehaus.org/display/JETTY/Continuations` .
[7]See `http://incubator.apache.org/ode/jacob.html` .

## 7. REFERENCES


[1] G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. PhD thesis, MIT, 1985.

[2] S. Balsamo, P. Inverardi, and C. Mangano. An approach to performance evaluation of software architectures. In *WOSP '98: Proceedings of the 1st international workshop on Software and performance*, pages 178–190, New York, NY, USA, 1998. ACM Press.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Professional, 2 edition, 2005.

[4] D. Box. Code name indigo: A guide to developing and running connected systems with indigo. *MSDN Magazine*, January 2004. `http://msdn.microsoft.com/msdnmag/issues/04/01/indigo/default.aspx`.

[5] O. Boxma and J. Cohen. The m/g/1 queue with heavy-tailed service time distribution. *IEEE Journal on Selected Areas in Communications*, 16(5):749–763, 1998.

[6] J. Cao, M. Andersson, C. Nyberg, and M. Kihl. Web server performance modeling using an m/g/1/k*ps queue. In *Telecommunications, 2003. ICT 2003. 10th International Conference on*, volume 2, pages 1501–1506 vol.2, 2003.

[7] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley Professional, 2000.

[8] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, August 2005.

[9] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[10] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Trans. Inter. Tech.*, 2(2):115–150, 2002.

[11] S. Haines. *Pro Java EE 5 Performance Management and Optimization*. Apress, 2006.

[12] J. Hasan and K. Tu. *Performance Tuning and Optimizing ASP.NET Applications*. Apress, 2003.

[13] J. John C. Sees and J. F. Shortle. Difficult queuing simulation problems: simulating m/g/1 queues with heavy-tailed service. In *WSC '02: Proceedings of the 34th conference on Winter simulation*, pages 433–438. Winter Simulation Conference, 2002.

[14] D. A. Manolescu. *Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[15] D. A. Manolescu. Workflow enactment with continuation and future objects. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 40–51, New York, NY, USA, 2002. ACM Press.

[16] J. McConnell and E. Siegel. *Practical Service Level Management: Delivering High-Quality Web-Based Services*. Cisco Press, 2004.

[17] R. Milner. *The polyadic pi-calculus: a tutorial*, pages 203–246. Springer-Verlag, 1993.

[18] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing execution of composite web services. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 170–187, New York, NY, USA, 2004. ACM Press.

[19] E. Newcomer and G. Lomow. *Understanding SOA with Web Services*. Addison Wesley Professional, 2004.

[20] OASIS. Reference model for service oriented architecture v 1.0. Web, July 2006. `http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf`.

[21] OASIS. Web services business process execution language version 2.0. Web, Feb 2007. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf`.

[22] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Trans. Softw. Eng.*, 21(8):689–700, 1995.

[23] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, 7 edition, 2004.

[24] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 291–302, New York, NY, USA, 2005. ACM Press.

[25] W3C. Web services addressing 1.0 - core. Web, May 2006. `http://www.w3.org/TR/2006/REC-ws-addr-core-20060509/`.

[26] W3C. Web services description language (wsdl) version 2.0 part 2: Adjuncts. Web, March 2006. `http://www.w3.org/TR/2006/CR-wsdl20-adjuncts-20060327/`.

[27] M. Welsh. *An Architecture for Highly Concurrent, Well-Conditioned Internet Services*. PhD thesis, University of California, Berkeley, August 2002.

[28] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM Press.