A QoS-Aware Fault Tolerant Middleware for Dependable Service Composition

Zibin Zheng and Michael R. Lyu
Department of Computer Science and Engineering
The Chinese University of Hong Kong
{zbzheng, lyu}@cse.cuhk.edu.hk

Abstract

Based on the framework of service-oriented architecture (SOA), complex distributed systems can be dynamically and automatically composed by integrating distributed Web services provided by different organizations, making dependability of the distributed SOA systems a big challenge. In this paper, we propose a QoS-aware fault tolerant middleware to attack this critical problem. Our middleware includes a user-collaborated QoS model, various fault tolerance strategies, and a context-aware algorithm in determining optimal fault tolerance strategy for both stateless and stateful Web services. The benefits of the proposed middleware are demonstrated by experiments, and the performance of the optimal fault tolerance strategy selection algorithm is investigated extensively. As illustrated by the experimental results, fault tolerance for the distributed SOA systems can be efficient, effective and optimized by the proposed middleware.

1. Introduction

Service-oriented architecture (SOA) is becoming a major software framework for distributed systems. In the service-oriented environment, complex distributed systems can be dynamically and automatically composed by integrating existing Web services, which are provided by different organizations. Since the Web service components are usually distributed across the Internet and invoked by communication links, building dependable SOA systems becomes a great challenge.

Software fault tolerance is an important approach for building reliable systems. One approach to software fault tolerance, also known as design diversity, is to employ functionally equivalent yet independently designed program versions [11]. This used-to-be expensive approach now becomes a viable solution to the fast-growing service-oriented computing arena, since the independently designed Web services with overlapping or identical functionalities are suited for the construction of diversity-based fault tolerant systems. There is an urgent need for systematic studies on how to apply traditional software fault tolerance techniques to the service-oriented computing arena.

Our work aims at advancing the current state-of-the-art in fault tolerance technologies for dependable service composition. We propose a QoS-aware fault tolerant middleware to make fault tolerance for the distributed SOA systems efficient, effective and optimized. The contributions of this paper are three-fold: 1) to comply with the key concept of Web 2.0, user-collaboration is introduced in our QoS model of Web services, and systematic formula and algorithms for QoS composition are provided; 2) commonly-used fault tolerance strategies for service composition are identified; and 3) an adjustable context-aware algorithm is designed for determining optimal fault tolerance strategy dynamically and automatically for both stateless and stateful Web services.

Our middleware places great emphasis on applying fault tolerance techniques for stateful Web services, which is more challenging since stateful Web services are much more complex than stateless Web services. Although the proposed middleware is restricted to the service-oriented environment, most of the proposed techniques can also be applied to other distributed computing platforms (e.g., DCOM and CORBA) and stand-alone systems.

The rest of this paper is organized as follows: Section 2 introduces the system architecture. Section 3 defines the QoS model and fault tolerance strategies. Section 4 designs optimization algorithms. Section 5 shows our implementation and experiments and Section 6 concludes the paper.

2. System Architecture

Before introducing the system architecture, we first explain some basic concepts as follows: 1) atomic services present self-contained Web services which provide services to users independently without relying on any other Web services; 2) composite services present Web services which provide services by integrating other Web services; 3) ser-
service community, which is also introduced in [2, 27], defines common terminologies that are followed by all participants, so that the Web services developed by different organizations have the same interfaces and can be dynamically replaced by other functionally equivalent Web service at runtime; 4) service plan, which is defined in Definition 1, is an abstract description of activities for the SOA systems.

Definition 1 A service plan SP is a triple \((T, P, B)\), where 
\(T = SLT \cup SFT\) is a set of stateless tasks (SLT) and stateful tasks (SFT), \(P\) is a set of settings in the service plan (e.g. probabilities of the branches and loops structures, partial merge parameter of the parallel structures), and \(B\) provides the structure information of the service plan, which can be specified by XML based languages, such as BPEL [12].

As the basic assumption of the work [1, 26, 27, 29], we also assume that for each task in a service plan, there are multiple functionally equivalent service candidates in the service community can be adopted to fulfill the task. This paper focuses on how to employ the non-functional performance of the candidates and the preference of service users for dynamic optimal fault tolerance strategy determination.

As shown in Fig. 1, the work procedures of our middleware are as follows: 1) a service user (usually the developer of the SOA system) defines a service plan, 2) the middleware obtains a list of candidates and their overall non-functional QoS performance for each task in the service plan from different service communities, 3) the algorithm \(FT-BABHEU\) determines optimal fault tolerance strategies for the tasks in the service plan, 4) the execution engine in the middleware executes the service plan by invoking Web services with the selected fault tolerance strategy, and 5) the QoS module records the QoS information of the invoked services and exchanges this information with the community coordinators for new overall QoS information of the Web services.

### 3. QoS Model and Fault Tolerance Strategies

#### 3.1. User-collaborated QoS Model

In the presence of multiple Web services with identical functionalities, Quality-of-Service (QoS) provides non-functional characteristics for the optimal Web service selection. Based on the investigations of [1, 13, 27], we identify the most representative quality properties in our user-collaborated QoS model for Web services as shown in the following.

1. **Availability (av)** \(q^1\): the percentage of time that a Web service is operating during a certain time interval.
2. **Price (pr)** \(q^2\): the fee that a service user has to pay for invoking a Web service.
3. **Popularity (po)** \(q^3\): the number of received invocations of a Web service during a certain time interval.
4. **Data-size (ds)** \(q^4\): the size of the Web service invocation response.
5. **Success-rate (sr)** \(q^5\): the probability that a request is correctly responded within the maximum expected time.
6. **Response-time (rt)** \(q^6\): the time duration between service user sending a request and receiving a response.
7. **Overall Success-rate (osr)** \(q^7\): the average value of the invocation success rate \(q^5\) of all service users.
8. **Overall Response-time (ort)** \(q^8\): the average value of the response-time \(q^6\) of all service users.

In our QoS model, \(q^1\)–\(q^4\) are the same for all the service users and are provided by the service providers. \(q^5\) and \(q^6\) are affected by the communication links and are measured by the service users. \(q^7\) and \(q^8\) are the average values of \(q^5\) and \(q^6\), respectively. They are provided by the service community coordinators. Different from other QoS models, we introduce the concept of user-collaboration for obtaining the overall QoS information \((q^7\) and \(q^8\)), which can be achieved by encouraging the service users to contribute their individually observed QoS information to the community coordinators for exchanging QoS information of other service users. The overall QoS properties provide critical data for the Web service selection, especially for the new service users, who have no knowledge on the performance of the functionally equivalent service candidates.

This QoS model is extensible, where more quality properties [19] can be added in the future without fundamental changes. Given the above quality properties, the QoS performance of a Web service can be presented as: \(q = (q^1, ..., q^8)\).
Table 1. Composition Formula for Basic Compositional Structures and Fault Tolerance Strategies

<table>
<thead>
<tr>
<th>QoS Properties</th>
<th>Basic Structures</th>
<th>Fault Tolerance Strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>$\sum_{i=1}^{n} q_i^t$</td>
<td>$\sum_{i=1}^{m} p_i q_i^t$</td>
</tr>
<tr>
<td>parallel</td>
<td>$\max q_i^t$</td>
<td>$\sum_{i=1}^{m} q_i^t$</td>
</tr>
<tr>
<td>branch</td>
<td>$\sum_{i=0}^{n} p_i q_i^t i$</td>
<td>$\sum_{i=1}^{m} p_i i q_i^t$</td>
</tr>
<tr>
<td>loop</td>
<td>$\sum_{i=1}^{n} p_i q_i^t i$</td>
<td>$\sum_{i=1}^{m} p_i \left( \sum_{j=1}^{i} q_j^t \right)$</td>
</tr>
<tr>
<td>retry</td>
<td>$\sum_{i=1}^{r} p_i q_i^t i$</td>
<td>$\max q_i^t$</td>
</tr>
<tr>
<td>rb</td>
<td>$\sum_{i=1}^{m} p_i \left( \sum_{j=1}^{i} q_j^t \right)$</td>
<td>$\min q_i^t$</td>
</tr>
<tr>
<td>nvp</td>
<td>$\sum_{i=1}^{m} q_i^t$</td>
<td>active</td>
</tr>
</tbody>
</table>

Figure 2. Basic Compositional Structures

3.2. QoS Composition

Atomic services can be aggregated by different compositional structures. Figure 2 shows the basic compositional structures for describing the order in which a collection of tasks is executed. In the branch-split structure, $P_1=\{p_i\}_{i=1}^{n}$ is a set of execution probabilities of different branches, where $\sum_{i=1}^{n} p_i = 1$. In the loop structure, $P_2=\{p_i\}_{i=0}^{n}$ is a set of probabilities of executing the loop for $i$ times, where $n$ is the maximum loop times and $\sum_{i=0}^{n} p_i = 1$. In the parallel-split structure, all tasks will be executed in parallel, so each branch has an execution probability of 1. The parallel-join supports partial-merge by the parameter $k$, which means that the following task $t_{n+1}$ will be executed only after the finish of $k(1 \leq k \leq n)$ or more than $k$ parallel tasks. The basic structures in Fig. 2 are included in BPMN [18] and can be mapped to BPEL [12] easily. We use these structures to model service compositions in this paper.

The QoS values of the composite services, which aggregate atomic services employing the basic compositional structures (sequence, parallel, branch and loop), can be calculated by the formula in Table 1. In the parallel structure, the response-time (rt) is the maximum value of the first $k$ returned parallel branches. The parallel structure is counted as a success if $k$ or more than $k$ branches success. $S^x(i)$ is designed for calculating the probability that $i$ parallel branches from all the $n$ branches success, where $x=1, 5, 7$. For example, when $n=3$, $k=2$, then $q^x=S^x(2)+S^x(3)$, where $S^x(2)=q_1^x q_2^x (1-q_3^x)+q_3^x (1-q_2^x)$ and $S^x(3)=q_1^x q_2^x q_3^x$.

The basic structures can be nested and combined in arbitrary ways. For calculating the aggregated QoS values of a service plan, we decompose the service plan to basic structures hierarchically using Algorithm 1. When a decomposed sub-service-plan is a basic structure, the formula in Table 1 are employed for calculating the QoS values. Then the QoS values of this sub-service-plan can be employed for calculating QoS values of its parental plans.

3.3. Fault Tolerance Strategies

To build dependable service-oriented systems, the functionally equivalent candidates in a service community can be employed as alternative replicas for tolerating faults. The commonly used fault tolerance strategies for service composition are identified in the following and the formula for calculating their QoS values are listed in Table 1.

- **Retry.** The original Web service will be tried for a certain number of times if it fails. In Table 1, $r \geq 2$ is the maximal execution times of the original task. $p_i$ is the probability that $t_i$ will be executed for $i$ times.
\( p_i \) can be calculated by 
\( p_i = (1 - q_{i1}^n)^{t-1} \times q_i^n \), where 
\( q_i^n \) is the success-rate of the target Web service.

- **Recovery Block (RB).** Another standby Web service will be tried sequentially if the primary Web service fails. In Table 1, \( m \) (\( m \leq \text{number of candidates} \)) is the maximal execution times, and \( p_i \) is the probability that the \( i^{th} \) candidate will be executed. \( p_i \) can be calculated by 
\( p_i = (1 - q_{i1}^n)^{t-1} \times q_i^n \).

- **N-Version Programming (NVP).** All the \( n \) functionally equivalent versions are invoked in parallel and the final result will be determined by majority voting.

- **Active.** All the \( n \) functionally equivalent versions are invoked in parallel and the first returned result without network errors will be selected as the final result.

For each abstract task in a service plan, there are two types of candidates can be adopted for implementing the task: 1) Atomic services without any fault tolerance strategies. 2) Composite service with fault tolerance strategies (Retry, RB, NVP and Active).

The selection algorithms proposed in Section 4 will be employed for optimal candidate determination. The fault tolerance strategies in the middleware can be easily replaced and updated, since the selection algorithm in Section 4 is independent of these strategies.

### 4. Fault Tolerance Strategy Selection

#### 4.1. Notations and Utility Function

The notations used in the following of this paper are defined in Table 2. Given a task \( t_i \), there is a set of candidates \( S_i \). Each candidate \( s_{ij} \) has a quality vector \( q_{ij} = \{q_{ij}^k\}_{k=1}^c \) presenting the nonfunctional characteristics, where \( c \) is the number of quality properties. Since some quality properties are positive (lager value for higher quality, such as availability and popularity) and some are negative (smaller value for better quality), we first transform all the positive quality properties to negative ones using \( \tilde{q}_{ij}^k = \max q_i^k - q_{ij}^k \), where \( \max q_i^k \) is the maximal value of all the candidates. Since different quality properties have different scales, we employ a Simple Additive Weighting (SAW) technique [4] to normalize the quality properties, which is defined as follows:

\[
\tilde{q}_{ij}^k = \begin{cases} 
q_{ij}^k - \min q_i^k & \text{if } \max q_i^k \neq \min q_i^k \\
1 - \min q_i^k & \text{if } \max q_i^k = \min q_i^k 
\end{cases} 
\]

\[ (1) \]

To calculate the performance of different candidates, a utility function is defined as:

\[
u_{ij} = \text{utility}(q_{ij}) = \sum_{k=1}^{c} w_k \times \tilde{q}_{ij}^k, \]

\[ (2) \]

Table 2. Notations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( SP )</td>
<td>a service plan, which is a triple ( T; F; B ).</td>
</tr>
<tr>
<td>( T )</td>
<td>a set of tasks in the service plan, ( T = SLT \cup SFT ).</td>
</tr>
<tr>
<td>( SLT )</td>
<td>a set of stateless tasks, ( SLT = {t_i}_{i=1}^{n_L} ).</td>
</tr>
<tr>
<td>( SFT )</td>
<td>a set of stateful tasks, ( SFT = {SFT_i}_{i=1}^{n_F} ).</td>
</tr>
<tr>
<td>( SFT_i )</td>
<td>a set of related tasks of the ( i^{th} ) stateful task.</td>
</tr>
<tr>
<td>( n )</td>
<td>the number of tasks in ( SP ), ( n = n_L + n_F ).</td>
</tr>
<tr>
<td>( n_L )</td>
<td>the number of the stateless tasks in ( SP ), ( n_L =</td>
</tr>
<tr>
<td>( n_F )</td>
<td>the number of the stateful tasks in ( SP ), ( n_F =</td>
</tr>
<tr>
<td>( n_i )</td>
<td>number of state related tasks of ( SFT_i ), ( n_i =</td>
</tr>
<tr>
<td>( S_i )</td>
<td>a set of candidates for ( t_i ), ( S_i = {s_{ij}}_{j=1}^{n_i} ).</td>
</tr>
<tr>
<td>( m_i )</td>
<td>the number of candidates for ( t_i ), ( m_i =</td>
</tr>
<tr>
<td>( LC )</td>
<td>the optimal candidate index for ( t_i ).</td>
</tr>
<tr>
<td>( LC_i )</td>
<td>local constraints for task ( t_i ), ( LC_i = {l_{ij}^k}_{j=1}^{m_i} ).</td>
</tr>
<tr>
<td>( GC )</td>
<td>global constraints for ( SP ), ( GC = {g_{ik}^k}_{k=1}^{c} ).</td>
</tr>
<tr>
<td>( e )</td>
<td>the number of quality properties.</td>
</tr>
<tr>
<td>( q_{ij} )</td>
<td>a quality vector for ( s_{ij} ), ( q_{ij} = {q_{ij}^k}_{k=1}^{c} ).</td>
</tr>
<tr>
<td>( ER )</td>
<td>a set of execution routes of ( SP ), ( ER = {ER_i}_{i=1}^{c} ).</td>
</tr>
<tr>
<td>( n_e )</td>
<td>the number of execution routes of a service plan.</td>
</tr>
<tr>
<td>( pro(ER_i) )</td>
<td>the execution probability of ( ER_i ).</td>
</tr>
<tr>
<td>( SR )</td>
<td>a set of sequential routes of ( SP ), ( SR = {SR_i}_{i=1}^{c} ).</td>
</tr>
<tr>
<td>( n_s )</td>
<td>the number of sequential routes of ( SP ).</td>
</tr>
<tr>
<td>( pct )</td>
<td>a user defined threshold for ( ER ).</td>
</tr>
</tbody>
</table>

where \( w_k \) is the user-defined weights for presenting the priorities of different quality properties and a smaller \( u_{ij} \) means better performance.

#### 4.2. FT Selection with Local Constraints

Local constraints, \( LC = \{l_{ij}^k\}_{k=1}^{c} \), specify user requirements for a single task in a service plan (e.g., response-time has to be smaller than 1 second). There are totally \( n \times c \) local constraints for a service plan, where \( n \) is the number of tasks and \( c \) is the number of quality properties. Usually, users only set a small subset. Since all the quality properties are transformed to negative, the untouched local constraints are set to \( +\infty \) by default, so that all the candidates meet the constraints.

The optimal candidate selection problem for a single stateless task \( t_i \) with local constraints can be formulated mathematically as Problem 1, where \( x_{ij} \) is set to 1 if the candidate \( s_{ij} \) is selected and 0 otherwise. In Problem 1, \( q_{ij} = \{q_{ij}^k\}_{k=1}^{c} \) is the quality vector of \( s_{ij} \), \( u_{ij} \) is the utility value of the candidate \( s_{ij} \) calculated by Equation 2, and \( m_i = |S_i| \) is the number of candidates for task \( t_i \).

**Problem 1** Minimize: 
\[ \sum_{j=1}^{m_i} u_{ij} x_{ij} \]

Subject to:

- \[ \sum_{j=1}^{m_i} q_{ij} x_{ij} \leq l_{ij}^k (k = 1, 2, ..., c) \]
- \[ \sum_{j=1}^{m_i} x_{ij} = 1 \]
Data: Service plan $SP$, local constraints $LC$, candidates $S$

Result: Optimal candidate index $\rho$ for $SP$.

$n_l = |SLT|$; $n_f = |SFT|$; $n_t = |SFT_i|$; $m_i = |S_i|$; $m_{ij}$

for ($i = 1; i \leq n_l; i++)$
  for ($j = 1; j \leq m_i; j++)$
    if $x_{ij} \leq c_{ij}$
      then $u_{ij} = utility(q_{ij})$;
  end
  end
if no candidate meet $lc$, then Throw exception;
end
for ($i = 1; i \leq n_f; i++)$
  for ($j = 1; j \leq n_t; j++)$
    if $x_{ij} \leq c_{ij}$
      then $q = f(lowQoS(SP, q_{ij}, ...q_{in}))$;
    $u_{ij} = utility(q)$;
  end
end
if no candidate meet $lc$, then Throw exception;
end
for all tasks in $SFT$, do $\rho_k = x$;
end

Algorithm 2: FT Selection with Local Constraints

- $x_{ij} \in \{0, 1\}$

To solve Problem 1, for each task $t_i$, we first use the formula in Table 1 to calculate the QoS values of the fault tolerance strategy candidates. Then the candidates which cannot meet the local constraints are excluded. After that, the utility values of the candidates are calculated by Equation 2. Finally, the candidate $s_{ij}$ with the best utility value will be selected as the optimal candidate for $t_i$ by setting $\rho_i = x$.

When a service plan contains stateful tasks and needs to maintain states across multiple tasks, the state-related tasks need to employ operations provided by the same Web service (e.g., we cannot login in one Web service and logout in another one). Algorithm 2 is designed to select optimal candidates for a service plan, which includes stateless tasks ($SLT = \{t_i\}_{i=1}^{n_l}$) as well as stateful tasks ($SFT = \{SFT_i\}_{i=1}^{n_f}$). Algorithm 2 first selects optimal candidates for the stateless tasks using the above procedures. Then, for each stateful task $SFT_i$, which includes a set of state-related tasks, the overall QoS values of the whole service plan with different candidate-sets (operations of the same Web service) are calculated by Algorithm 1, and the utility values are calculated by Equation 2. Finally, the candidate-set which meets all the local constraints and with the best utility value will be selected as the optimal candidate-set for $SFT_i$.

4.3. FT Selection with Global Constraints

Local constraints require service users to provide detailed constraint settings for individual tasks in the service plan, which not only needs a lot of time for the configuration, but also requires good knowledge on the individual tasks. To address these drawbacks, we design global constraints (GC) for specifying constraints for the whole service plan. For a service plan, there are a set of global constraints $GC = \{gc\}_{i=1}^{n_c}$ for the $c$ quality properties respectively.

Since a service plan may include branch structures and has multiple execution routes, each execution route should meet the global constraints to make sure that the whole service plan meets the global constraints. The execution route and sequential route are defined as follows:

Definition 2 Execution route ($ER_i$) is a sub service plan ($ER_i \subseteq SP$) including only one branch in each branch structure. Each execution route has an execution probability $pro(ER_i)$, which is the product of all probabilities of the selected branches in the route. $\sum_{i=1}^{n_c} pro(ER_i) = 1$, where $n_c$ is the number of execution routes in a service plan.

Definition 3 Sequential route ($SR_{ij}$) is a sub service plan which includes only one branch in each parallel structure of an execution route, $SR_{ij} \subseteq ER_i$.

For determining optimal candidates for an execution route under global constraints, the simplest way is employing an exhaustive searching approach to calculate utility values of all candidate combinations and select out the one, which meets all the constraints and with the best utility performance. However, exhaustive searching approach is impractical when the task number is large, since the number of candidate combinations $\prod_{i=1}^{n_c} m_i$ is increasing exponentially, where $m_i$ is the candidate number for task $t_i$ and $n_c$ is the task number in the service plan.

To determine the optimal fault tolerance candidates for a service plan under both global and local constraints, we first transform the loop structures to branch structures using the approach proposed in [1], where the $i$th branch presents executing the loop for $i$ times. Then, a service plan is decomposed to different execution routes, and for each execution route, the optimal candidate determination problem is modeled as a 0-1 Integer Programming (IP) problem as shown in Problem 2.

Problem 2 Minimize:

$$\sum_{i \in ER_j} \sum_{j \in S_i} u_{ij}x_{ij}$$ (3)

Subject to:

$$\sum_{i \in ER_j} \sum_{j \in S_i} q_{ij}x_{ij} \leq gc^0 (y = 2, 3, 4)$$ (4)

$$\forall k, \sum_{i \in SR_{ik}} \sum_{j \in S_i} q_{ij}x_{ij} \leq gc^k (y = 6, 8)$$ (5)

$$\prod_{i \in ER_j} \prod_{j \in S_i} (q_{ij})^{y_{ij}} \leq gc^y (y = 1, 5, 7)$$ (6)

$$\forall SFT_i, x_{y_{1j}} = x_{y_{2j}} = ... = x_{y_{nj}} (l_{yi} \in SFT_i)$$ (7)
∀i, ∑ j∈Si xij = 1; xij ∈ {0, 1} (8)

In Problem 2, Equation 3 is the objective function, where \( u_{ij} \) is the utility value of the candidate \( s_{ij} \). Equation 4 is the global constraints for the quality properties price, popularity and date-size \((q^p, y = 2, 3, 4)\), where the QoS values of the whole execution route are the sum of all its tasks. Equation 5 is the global constraints for Response-time and overall-response-time \((q^o, y = 6, 8)\). For \( q^6 \) and \( q^8 \), all sequential routes in the execution route should meet the global constraints \( gc^6 \) and \( gc^8 \) to make sure that the response time of the longest sequential route meets the global constraints. The QoS values of \( q^6 \) and \( q^8 \) of the sequential-routes are calculated by the sum of all its tasks. Equation 6 is the global constraints for availability, success-rate and overall-success-rate \((q^a, y = 1, 5, 7)\), where \( (q^a_{\text{pro}})_{x_{\text{ij}}} = 1 \) if a candidate is not selected \((x_{\text{ij}} = 0)\). The QoS values of \( q^1 \), \( q^7 \), and \( q^8 \) can be calculated by the product of the tasks. Equation 7 is to make sure that the state-related tasks in \( SFT_i \) will employ operations of the same Web service (the same candidate index \( j \)). Equation 8 is to make sure that only one candidate will be selected for each task.

In integer programming, the objective function and constraints should be linear. Therefore, we need to transform the Equation 6 from non-linear to linear. By applying the logarithm function to Equation 6, we obtain

\[
\sum_{i \in ER_i} \sum_{j \in S_i} x_{ij} \ln(q^i_{\text{pro}}(y = 1, 5, 7)),
\]

which is linear. The objective function need to be changed accordingly. When calculating the QoS values \((q^p, y = 1, 5, 7)\) of the execution route, the normalization function \( q^{\text{max}} - q^{\min} \) should be replace by

\[
\frac{\tilde{q}^{y}_{\text{ER}_i} - \min \ln(q^{y})}{\max \ln(q^{y}) - \min \ln(q^{y})},
\]

where

\[
\tilde{q}^{y}_{\text{ER}_i} = \ln(q^{y}_{\text{ER}_i}) = \sum_{i \in ER_i} \sum_{j \in S_i} x_{ij} \ln(q^i_{\text{pro}}).\]

In this way, the optimal fault tolerance strategy determination problem is formulated as a 0-1 IP problem. Then, we design an algorithm \( FT-BAB \), which is based on the well-known Branch-and-Bound algorithm [23], to find optimal fault tolerance strategies for the execution routes. Since each execution route may only include a subset of the whole service plan and different execution routes may have overlapping tasks, the following rules are designed to combine the results:

**Algorithm 3: Hybrid Algorithm: FT-BABHEU**

- If a task \( t_i \) only belongs to one execution route, then the optimal result is selected as the final result for the service plan.
- If a task \( t_i \) belongs to multiple execution routes, then the result of the execution route with the highest execution probability \( \text{pro}(ER_i) \) is selected as the final result for the service plan.

4.4. Hybrid Algorithm for FT Selection

The IP problem is NP-Complete [6] and the computation time increases exponentially with the problem size. To address these drawbacks, we design a hybrid algorithm \( FT-BABHEU \) as shown in Algorithm 3, which combines a Branch-and-Bound algorithm \( FT-BAB \) and a Heuristic algorithm \( FT-HEU \) for improving the computation performance. The parameter \( pct \) (0% ≤ \( pct \) ≤ 100%) in line 3 of the Algorithm 3 is a user-defined threshold for adjusting the \( FT-BABHEU \) algorithm. An execution route is counted as a major route if its execution probability \( \text{pro}(ER_i) \) is in the top \( pct \) percent of all the execution routes. When \( pct = 100\% \), all execution routes are major routes and when \( pct = 0\% \), there are no major routes. Algorithm 3 includes the following main steps:

**Step 1** (lines 2-7): Finding out the optimal candidates for the major routes by solving the IP problem using a Branch-and-Bound algorithm \( FT-BAB() \).
Data: SP, GC, LC, S, Te, ρ
Result: Initial candidates index ρ for SP.

\[ q_{all} = \text{flowQoS}(SP, q_{i1}, \ldots, q_{ir_{ij}}); \]

\[ \frac{\rho_{\text{old}}}{\rho_{\text{new}}} \leq \frac{\rho_{\text{old}}}{\rho_{\text{new}}}; \]

Algorithm 4: Find Initial Solution

**Step 2** (lines 8-15): Combining the optimal results of different execution routes by the rules in Section 4.3.

**Step 3** (lines 16-17): If the major routes cover all the tasks in the service plan, the optimal results will be returned. Otherwise, a heuristic algorithm FT-HEU will be employed for determining the optimal candidates for the uncovered tasks. In the FT-HEU algorithm, first the function findInitialSolution(), which is shown in Algorithm 4, is invoked for finding initial feasible candidates for the uncovered tasks. For each candidate of an uncovered task, Equation 12 is employed for calculating the value of λij, where a smaller value means the candidate is more suitable. q^{k}_{all} in Equation 12 is the accumulated QoS values of all the selected candidates, which can be calculated by Algorithm 1. When the value of \( \frac{q^{k}_{all}}{g^{k}_{c}} \) is large, it means that the quality property \( g^{k}_{c} \) is in more danger and needs more attention (larger \( w_{k} \)).

\[ \lambda_{ij} = \sum_{t=1}^{c} w_{k} q^{k}_{all} / \sum_{t=1}^{c} g^{k}_{c}; \]  

\[ w_{k} = \begin{cases} \frac{1}{c} & \text{if } q_{all} = 0; \\ \frac{\rho_{\text{old}}}{\rho_{\text{new}}} / \sum_{k=1}^{c} \frac{q^{k}_{all}}{g^{k}_{c}} & \text{if } q_{all} \neq 0 \end{cases} \]  

(12)

**Step 4** (lines 18-26): If the initial solution can not meet the global constraints (denoted as infeasible solution), then the findExchangeCandidate() function, which is shown in Algorithm 5, is invoked to find an exchangeable candidate which meets the following three requirements:

- It will decrease the highest infeasible factor of the quality properties, \( \frac{q^{\text{new}}_{\text{all}}}{g^{\text{c}}_{\text{new}}} < \frac{q^{\text{old}}_{\text{all}}}{g^{\text{c}}_{\text{old}}} \), where \( \frac{q^{\text{old}}_{\text{all}}}{g^{\text{c}}_{\text{old}}} = \max(\frac{q^{k}_{\text{all}}}{g^{k}_{c}}, \ldots, \frac{q^{k}_{\text{all}}}{g^{k}_{c}}) \) and \( \frac{q^{\text{new}}_{\text{all}}}{g^{\text{c}}_{\text{new}}} > 1 \).
- It will not increase the infeasible factor of any other previously infeasible properties, \( \forall y(\frac{q^{\text{new}}_{y}}{g^{y}_{c}} \leq \frac{q^{\text{old}}_{y}}{g^{y}_{c}}) \), where \( \frac{q^{\text{old}}_{y}}{g^{y}_{c}} > 1 \) and \( y \neq x \).
- It will not make any previously feasible quality properties become infeasible, \( \forall y(\frac{q^{\text{new}}_{y}}{g^{y}_{c}} \leq 1) \), where \( \frac{q^{\text{old}}_{y}}{g^{y}_{c}} \leq 1 \).

If such a candidate cannot be found, then a FeasibleSolutionNotFound exception will be thrown to the user for relaxing the constraints. Otherwise, the above candidate-exchanging procedures will be repeated until a feasible solution becomes available.

**Step 5** (lines 27-29): Iterative improvement of the feasible solution by invoking the feasibleUpgrade() function, which is shown in Algorithm 6. The feasible solution upgrade includes the following steps:

- If there exists at least one feasible upgrade (smaller utility value \( u_{\text{new}} < u_{\text{old}} \)) which provides QoS savings \( v_{ij} < 0 \), the candidate with maximal QoS savings (minimal \( v_{ij} \) value) is chosen for exchanging. The QoS saving \( v_{ij} \) is defined as:

\[ v_{ij} = \sum_{k=1}^{c} w_{k} \frac{q^{\text{new}}_{k} - q^{\text{old}}_{k}}{g^{k}_{c}}; \]  

(13)

where \( w_{k} \) is defined in Equation 12.
- If no feasible upgrade with QoS saving exists, then the candidate with maximal utility-gain per QoS saving is selected, which is calculated by \( \frac{\rho_{\text{old}} - \rho_{\text{new}}}{v_{xy}} \).

Algorithm 5: Find Exchange Candidate
Data: Service plan SP, Constraints GC, LC, Candidates S
Result: Candidates indexρ for SP.

$$n = |SLT| + |SFT|, n_l = |SLT|, n_{all} = |S|;$$

$$q_{old} = \text{flowQoS}(SP, q_{i1}, \ldots, q_{i\varphi});$$

$$u_{old} = \text{utility}(q_{old});$$

$$w_i = \begin{cases} \frac{u_{old} - u_{all}}{n_{all}} & \text{if } q_{all} = 0 \\ \frac{u_{all} - u_{old}}{n_{all}} & \text{if } q_{all} \neq 0 \end{cases};$$

for (i\text{=1; } i \leq n; i++) do
  for (j\text{=}i+1; j \leq n; j++) do
    if \text{findExchangeCandidate}\_\text{Index}(i, j) = 0 then 
      \text{Continue;}
    end

end

if \text{exists}(u_{xy} < u_{old} \& \& u_{xy} < 0 \& \& v_{ij} < 0) then
  \rho_x = y;
else
  \rho_x = \text{y};
end

return \rho.

Algorithm 6: Feasible Upgrade of the Solution

The FT-HEU algorithm has convergence property, since
1) Step 4 never makes any feasible properties to become
infeasible or infeasible properties to be more infeasible, and
for each exchange, the property with the maximal infeasible
factor will be improved; 2) Step 5 always upgrades the util-
ity value of the solutions. Because there are only a finite
number of feasible solutions, the algorithm cannot cause
any infinite looping.

For calculating the upper bound of the worst-case compu-
tational complexity of the FT-HEU algorithm (\(\text{pct} = 0\%\)), we assume there are \(n\) tasks, \(m\) candidates for each
and \(c\) quality properties. In Step 3, when finding the
initial solution (Algorithm 4), the computation of \(\lambda_{ij}\)
is \(O(nm)\). In Step 4, finding a exchange candidate (Algo-
rium 5) requires a maximum of \(n(m-1)\) of calcu-
lation of the alternative candidates, and each calculation
will invoke a function \(\text{flowQoS}\), which has the compu-
tation complexity of \(O(nc)\). Therefore, the compu-
tational complexity is \(O(n^2(m-1)c)\) of each exchange. The
\text{findExchangeCandidate()} function will be invoked at
most \(n(m-1)\) times since there are at most \((m-1)\)
upgrades for each task. Therefore, the total computation
complexity of Step 4 is \(O(n^3(m-1)^2c)\). In Step 5, for
each upgrade, there are \((n-1)\) iteration for the alter-
native candidates and for each iteration. For each iteration,
the \(\text{flowQoS}\) function, which has complexity \(O(nc)\), is
invoked. So the computation complexity of each upgrade is
\(O(n^2(m-1)c)\). There are totally \(n(m-1)\) upgrades for
the whole service plan, so the total computation complex-
ity of Step 5 is \(O(n^3(m-1)^2c)\). Since Step 3, Step 4 and
Step 5 are in sequence, thus the combined complexity of
the \(\text{FT-HEU}\) algorithm is \(O(n^3(m-1)^2c)\).

5. Implementation and Experiments

To study the performance of different selection algo-
rithms (\(\text{FT-Local}, \text{FT-ALL}, \text{FT-BAB}, \text{FT-HEU}, \text{FT-BABHEU}\)
), we use an Internet topology generator \text{Inet 3.0} \cite{9}
to create 10000 random nodes for presenting different Web
services in the Internet. We then randomly select differ-
ent number of nodes to create service plans with differ-
ent compositional structures and execution routes. The
\(\text{FT-Local}\) algorithm presents the selection algorithm with
local constraints proposed in Algorithm 2, the \(\text{FT-ALL}\)
presents the exhaustive searching approach introduced in
Section 4.3, the \(\text{FT-HEU}\) presents the heuristic algorithm
(\(\text{pct} = 0\%\)), \(\text{FT-BAB}\) presents the Branch-and-Bound
algorithm (\(\text{pct} = 100\%)\) for solving the IP problem, and
\(\text{FT-BABHEU}\) presents the hybrid algorithm shown in Algo-
rithm 3. All the algorithms are implemented in the Java lan-
guage and the \text{LP-SOLVE} package (lpsove.sourceforge.net)
is employed for the implementation of the \(\text{FT-BAB}\) algo-
rithm. The configurations of the computers for running the experiments are: \text{Intel(R) Core(TM)2 2.13G CPU with 1G
RAM, 100Mbits/sec Ethernet card, Window XP and JDK
6.0}. In the following, we present the experimental results of
\text{computation time and selection results}.

5.1. Computation Time

Figure 3(a), (b), and (c) shows the computation time per-
formance of different algorithms with different number of
the tasks, candidates and QoS properties, respectively. The
experimental result shows: 1) the computation time of \(\text{FT-
ALL}\) increase exponentially even with very small problem
size; 2) the computation time of FT-BAB is acceptable when the problem size is small, however, it increases quickly when the number of tasks, candidates and QoS properties is large; 3) the computation time of FT-HEU is very small in all the experiments even with large problem size; 4) the computation time performance of FT-Local is the best (near zero), however, FT-Local can not support global constraints. Figure 3(d) shows the computation time performance of FT-BABHEU with different \(\text{pct} \) settings. Figure 3(d) shows that the computation performance of FT-BABHEU is influenced by \(\text{pct} \), indicating that by setting the \(\text{pct} \) parameter, the FT-BABHEU algorithm can adapt to different environments.

5.2. Selection Results

Figure 4 compares the selection results of the FT-BAB and FT-HEU algorithms with different number of tasks, candidates and QoS properties. The y-axis of the Figure 4 is the values of \(\text{Utility(IP)/Utility(HEU)}\), which are the utility ratios of two algorithms, where the value of 1 means the selection results by FT-HEU is identical to the optimal result obtained by the FT-BAB.

Figure 4 (a) and (b) show the experimental results of FT-BAB and FT-HEU with different number of tasks and candidates, respectively. The experimental results show that: 1) under different number of QoS properties (10, 20, 30 and 40 in the experiment), the utility values of FT-HEU are near FT-BAB (larger than 0.975 in the experiment) with different number of tasks, candidates; 2) with the increasing of the task number, the performance of FT-HEU becomes better.

Figure 4(c) shows the selection result of FT-BAB and FT-HEU with different number of QoS properties. The result shows that the performance of FT-HEU is steady with different number of QoS properties in the experiments. Figure 4(d) shows the utility ratios of the FT-BABHEU with different \(\text{pct} \) settings. The experimental results show that 1) the selection results are influenced by the values of \(\text{pct} \), indicating that by setting the value of \(\text{pct} \), we can adjust the selection results of the FT-BABHEU algorithm; 2) when \(\text{pct} = 0\%\), the utility ratio is still larger than 99%, indicating the performance of the FT-HEU algorithm.

The above experimental results show that the FT-HEU algorithm can provide near optimal solutions with excellent computation time performance even under large problem size. By combining the accuracy feature of the FT-BAB algorithm and the speediness feature of the FT-HEU algorithm, our FT-BABHEU algorithm is adjustable and can be employed in different environments, such as the real-time applications (require quick-response), mobile Web services (limited computation resource), and large-scale service-oriented systems (large problem size). The design of the parameter \(\text{pct} \) in FT-BABHEU makes fault tolerance strategy personalization become easy (e.g., small \(\text{pct} \) for quick response and large \(\text{pct} \) for accurate selection results).

6. Discussion and Related Work

A number of fault tolerance strategies for Web services have been proposed in the recent literature [5, 8, 15, 21, 25, 30]. The major approaches can be divided into two types: 1) sequential strategies, where a primary service is invoked to process the request and the backup services are invoked only when the primary service fails. Sequential strategies have been employed in FT-SOAP [7] and FT-CORBA [24]; 2) parallel strategies, where all the candidates are invoked at the same time. Parallel strategies have been employed in FTWeb [22], Thema [14] and WS-Replication [20]. In this paper, we not only provide systematic introduction on the commonly-used fault tolerance strategies, but also propose a scalable middleware framework for dynamic fault tolerance strategy reselection and reconfiguration to deal with the frequently context information changes.

Recently, dynamic Web service composition has attracted great interest, where complex applications are specified as service plans and the optimal service candidates are dynamically determined at runtime by solving optimization problems. Although the problem of dynamic Web service selection has been studied by a number of literature [1, 3, 26, 27, 28], very few previous work focuses on the problem of dynamic optimal fault tolerance strategy determination, especially for stateful Web services. In this paper, we address this problem by proposing a hybrid algorithm FT-BABHEU, which is adjustable and can adapt to different environments easily.

The WS-Reliability [17] can be employed in our middleware for enabling reliable communication. WSRF [16], which describes the state as XML datasheets, can be employed for transferring states between replicas. The proposed middleware can be integrated into the SOA run-
time governance framework [10] and applied to industry projects.

7. Conclusion

In this paper, we have provided a practical solution for building dependable service-oriented systems by proposing a QoS-aware fault tolerant middleware. The main features of this middleware are: 1) supporting stateful Web services, 2) user-collaborated QoS model, 3) scalable middleware framework design to make replacement of the QoS properties and fault tolerance strategies easily, 4) the combination of the global constraints and local constraints for specifying user requirements, 5) a context-aware algorithm for dynamically and automatically optimal fault tolerance strategy determination.

Our future work will consider the state synchronization between different functionally equivalent Web services, the dependability guarantee of the middleware, and the investigation of more QoS properties.

Acknowledgement

The work described in this paper was fully supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK4158/08E), and a grant from the Research Committee of The Chinese University of Hong Kong (Project No. CUHK3/06C-SF).

References