AN APPROACH TO CONSTRUCTING HIGH-AVAILABLE DECENTRALIZED SYSTEMS VIA SELF-ADAPTIVE COMPONENTS*

XI SUN*, LI ZHOU†, LEI ZHUANG‡, WENPIN JIAO§ and HONG MEI¶

Software Institute, School of Electronics Engineering and Computer Science, Peking University
Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
Beijing 100871, P. R. China
*sunxi@sei.pku.edu.cn
†zhouli04@sei.pku.edu.cn
‡zhuanglei04@sei.pku.edu.cn
§jwp@sei.pku.edu.cn
¶meih@sei.pku.edu.cn

In decentralized computing environments, systems are built mainly from components that are developed and maintained independently by different third-party providers. The executions and evolutions of components located on distributed sites are beyond the control of the system developers, and the availabilities of those components are, to some extent, unpredictable because of their own tendencies and the unstable network. As a result, it is still a great challenge to construct high-available decentralized systems. In this paper, a self-adaptive component model is proposed to model those components distributed on the Internet and a running framework is described for constructing systems composed of self-adaptive components. Self-adaptive components can adjust their knowledge about the availabilities of the required services via learning from the feedback of historical invocations. Based on the knowledge, components can find the most appropriate service providers effectively and automatically. Experiments show that systems can always gain high availabilities under dynamic decentralized environments by using the approach.

Keywords: Self-adaptive component; decentralized system; availability.

1. Introduction

The emergence and popularization of new computing paradigms such as pervasive computing, grid computing, and service computing, make today’s software to be

*A preliminary version of this paper was presented at the 19th International Conference on Software Engineering and Knowledge Engineering (SEKE 2007) [16].

§Corresponding author.
increasingly “internet-scale” and “service-oriented”. Many internet-based applications reuse different third-party components to improve their development productivity and quality. We call these systems decentralized systems [1], which represent systems that are built mainly from components that are developed and maintained independently by different providers, on hardware nodes that are not under the control of the system developers. As more and more services and decentralized systems will certainly be deployed in the coming years, how to construct high-available decentralized systems is becoming a crucial challenge.

System availability can be defined as the degree to which a system is operational and accessible when required for use [2]. When situated in an unstable network environment, decentralized system’s availability is always weakened by the underlying connectivity failures. Moreover, third-party services often evolve outside of the systems, join or quit the network without notifying others. As a result, service availability often keeps on changing at runtime and, to some extent, is unpredictable. On the other hand, the proliferation of services available on the Internet increases the candidate providers. To improve the overall system availability, research efforts [3, 4] have been put into clustering many function-identical services and binding most adequate service(s) automatically during execution time.

Unfortunately, most existing solutions are using a centralized architecture style, which may be unsuitable in a large, decentralized environment with a large number of independent and distributed components. In the web service community, Ran [5] presents an approach to storing quality information (e.g., availability) of web services into the UDDI (Universal Discovery, Description, and Integration) registry. However, service availability depends on not only the service implementation but also the network status between service requestors and providers, and a service highly available to the UDDI server may provide a poor user-experience to the actual client. Another typical approach is to delegate customer requests to a service broker [6]. However, this approach needs the central broker to process a large number of client requests.

To cope with the uncertainty and dynamics in decentralized environments without a global coordinator, system components are required to approximate the current status of remote resources [1] (i.e., software entities constructing Internet-scale applications). That is, components should estimate the availabilities of service providers, and adjust the estimated values according to the service evolutions. Consequently, two important problems should be addressed at least: (1) given a collection of third-party services, how can the currently most available candidate be found effectively without a central information service? (2) When the quality of service or network connectivity changes, how can the system be aware of and adapt to the change automatically?

To address the above problems, in this paper, we present a component framework to facilitate constructing high-available self-adaptive systems in the decentralized environment. The main contributions of this paper are: First, we propose a self-adaptive component model to encapsulate the availabilities of remote components
and resources as adaptable “knowledge”. This model allows a component to dynamically estimate the availabilities of remote entities at runtime to adapt to the dynamic environment. Second, we design and implement an algorithm to reason about the availabilities of required services via learning from the feedback of historical invocations. This algorithm enables components to update their knowledge automatically. Finally, a running infrastructure is implemented to increase the feasibility of our approach.

The reminder of this paper is organized as follows. Section 2 proposes the self-adaptive component model, and describes the process of constructing systems based on self-adaptive components. Section 3 presents the algorithm for dynamically estimating and adjusting the availabilities of services in detail. Section 4 reports the experimental studies of the algorithm and analyzes the results. Section 5 describes the implementation details of the component framework. Section 6 compares our approach with some related work. Finally, Sec. 7 concludes this paper and discusses our future work.

2. Constructing High-Available Systems via Self-Adaptive Components

In this section, we first give a definition to the self-adaptive component. Then, we illustrate the runtime structure of the components in our current implementation. We finally describe the system constructing process.

2.1. Self-adaptive component definition

First, the definitions of interface and service are provided for completeness.

An interface is defined by Szyperski as:

**Definition 1.** An interface is a set of named operations that can be invoked by the clients [7].

We define a service as a program that implements and publishes one or more interfaces. A service may also depend on zero or more required interfaces.

**Definition 2.** A service is defined as a triple as follows:

\[ s = (\text{name}, \Pi_s, \Re_s) \]

- name is the unique name of service \( s \);
- \( \Pi_s \) is the set of provided interfaces of \( s \);
- \( \Re_s \) is the set of required interfaces of \( s \);

A self-adaptive component in our approach encapsulates the availabilities of its dependencies (a set of services that implement the component’s required interfaces) as adaptable “knowledge”. At runtime, the self-adaptive component selects the most available service from the candidates based on its knowledge, and adjusts its knowledge to adapt to the environment changes.
Definition 3. A self-adaptive component can be formally defined as a tuple as follows:

\[ c = (\text{name}, \text{PIs}, \text{RIs}, \text{D}, \text{Beh}) \]

where

- \text{name} is the unique name of component \( c \);
- \text{PIs} is the set of provided interfaces of \( c \);
- \text{RIs} is the set of required interfaces of \( c \);
- \( \text{D} \) is a relation to specify \( c \)'s dependencies, where: \( \text{D} = \text{RIs} \times (\text{S} \to \mathbb{R}) \). That is to say, for every required interface, there exist one or more service candidates that implement this interface (\( \text{S} \) is the set of the services). Moreover, for every service, component has a variable (which is within the range of \( \mathbb{R} \), the set of real numbers) to record the estimated value of the service’s availability.
- \text{Beh} is the component’s behavior to maintain the component’s dependencies and to adjust those estimated values related to the depended services’ availabilities. The component behavior is realized via an algorithm, which will be explained in depth in Sec. 3.

2.2. Runtime structure of self-adaptive components

When developing a self-adaptive component, developers should provide an implementation of the provided interfaces and a specification of the required interfaces separately. In the required interfaces specification, each service candidate should be specified with service type (e.g., web services, RESTful services [8]), address (e.g., URL), communication protocol (e.g., SOAP [9]), message structure, etc.

Figure 1 shows a runtime self-adaptive component from the structural perspective. The component is running in a container, and the service candidate list of every required interface is stored in the container’s dependency list. The component’s invocations to the required interfaces are delegated to the container’s selector module, which chooses one of the candidates and then requests the service via the
candidate’s connector. Our current implementation provides a tool to assist in generating the web service connector (i.e., the invocation stub) according to the given web service specification.

As shown in Fig. 2, briefly, there are three phases when a self-adaptive component invokes a service:

1. At the initialization stage, for each required interface, the self-adaptive component loads the list of service candidates that implement this interface. Then, the component designates a selection probability to each service candidate. Each candidate is initially assigned with the same probability.
2. When the component needs to invoke a service, the selector module chooses a service candidate according to the current specified probability distribution. (Steps 1–3 in Fig. 2)
3. After the invocation, the component adjusts the selection probabilities related to the service candidates according to the invocation feedback event (Steps 4–6 in Fig. 2). The adjustment is determined by the learning strategy module which uses the algorithm proposed in Sec. 3.

### 2.3. Self-adaptive component based systems

Self-adaptive component-based systems are composed of many self-adaptive components and service candidates. Note that a self-adaptive component is also a service from the definitions in Sec. 2.1; thus, a self-adaptive component can be used as a service candidate of other self-adaptive components.

A decentralized system could be a *closed* system, where each self-adaptive component uses a static list of service candidates. In our current implementation, developers can specify a component’s service candidates in a configuration file at the design phase. However, a decentralized system is usually an *open* system, where components can join and quit the system independently, thus the set of service candidates that a self-adaptive component can exploit changes continuously. In this section, we will focus on the open systems’ construction process.
2.3.1. Component initialization

The open decentralized systems are composed of the services which enter or quit the environment dynamically. Consequently, self-adaptive components cannot specify their depended services statically at the design phase. Thus, before running in the system, the self-adaptive components need to discover the services which implement the functionalities required by them in the current environment. Moreover, when a service enters or quits the system, the related self-adaptive components should be aware of this change.

The service discovery mechanism used in the open self-adaptive component-based systems borrows the idea of peer discovery in some hierarchical P2P systems, such as KaZaA [17] and eDonkey [18]. For each system, one or more system components act as registry services, which are responsible for processing the service publications and subscriptions of components. To achieve this, a registry service provides the participating components with a set of meta-operations, such as joining/leaving the system, publishing service descriptions, and subscribing published services. For normal services, a registry service behaves like a traditional service registry (e.g. the UDDI service); for self-adaptive components, a registry service notifies the registered component continuously about the registrations and departures of the services that the registered component has subscribed.

Figure 3 illustrates the detail steps in the component initialization. When a self-adaptive component (A2 in Fig. 3) joins the system, it first registers itself to a registry service it knows, and submits its service requirements; the registry service then returns a list of candidates that can provide the required services, besides, it also returns a list of registry services in the system. After that, the register service notifies the components which subscribe to the service provided by the newcomer, and broadcasts the information about the registered component to other registry services. The received service candidates are stored into the self-adaptive

![Diagram](image-url)

Fig. 3. Self-adaptive component initialization.
component’s dependency table, and the following interactions will occur between
the component and the selected service directly.

To mitigate the single-point-of-failure problem, more than one registry services
are deployed in the system, and they will share the received information as men-
tioned. A system component can reconnect to another registry service in its received
registry list if it cannot access its original register service. To make the system scal-
able and reliable, we adopt a hybrid solution to address the service entering/quitting
problem. On one hand, when a service wants to leave the system, it can notify its
registry service actively, then the registry service will notify the related self-adaptive
components and other registry services about this change; on the other hand, if the
service leaves without notifying its registry service or an error occurs in the network,
the self-adaptive components can also be aware of this change themselves with the
help of the learning algorithm proposed in Sec. 3. Thus, the registry services do not
have to keep checking the joined services’ states periodically. Moreover, the registry
services do not participate in the interactions of functional components and do not
have to process the quality-of-service related issues. Consequently, a registry service
would not become a bottleneck in a large-scale system.

2.3.2. System execution

In decentralized environments, services are running independently and continuously.
Every service processes requests from many different requestors, and may send
requests to other services. Furthermore, none of the services has a global view of
the system: each self-adaptive component only knows some dependent services, in
other words, a part of the system architecture. Figure 4 shows these components’
local views of the environment.

During the execution time, which services participate in the current computation
task is dependent on the self-adaptive components’ selection results. As shown in
Fig. 4, a request for service to component A may lead to the invocation to component

![Fig. 4. Self-adaptive component based system.](image-url)
B during A’s execution, and the path labeled with ‘a’ is the whole service process; to process another request, A may select component C and the process may be the path labeled with ‘b’ in the figure. Although the selection is dynamic, as the components process more invocations, the learning algorithm will enable them to find the most available service providers, and then the service process is increasingly stable.

In other words, the construction of a decentralized system is determined at runtime. Self-adaptive component-based systems are represented as coalitions of distributed entities which are self-organized to guarantee the high availability.

3. Estimated Service Availability Adjustment Algorithm

Our adjustment algorithm borrows the idea of the simulated annealing algorithm [10], but has many important differences in addressing the requirements of finding the service with highest availability in dynamic environments.

The estimated availability value $v_i$ of service candidate $i$ is calculated in Eq. (1) below.

$$v_i = e^{\alpha \times \text{history}}$$

There are two variables in Eq. (1), $\text{history}$ and $\text{temperature}$. Variable $\text{history}$ denotes the historical success rate of service candidate $i$. The success rate is quantified as the ratio of the number of successful completed interactions between the component and the candidate $i$ to the total number of attempted requests between them. A candidate who performs better in the history will have a bigger probability to be chosen in the next invocation.

We enlarge the influence of $\text{history}$ by multiplying it by a constant $\alpha$, which is designated to 60.0 in our current implementation. The value of $\alpha$ actually represents the degree of “confidence” about using the service candidates’ historical records to predict their future states. It should be different according to the environment where the component is situated. That is, in a stable environment, the value should be enlarged because the service availability does not change frequently (so that it is more predictable); on the contrary, if the environment is highly dynamic or unstable (e.g. many invocations sent to the candidates failed due to underlying connectivity failures), the value should be reduced. In the experiments we find the value 60.0 is appropriate according to our network environment. In our future work, we plan to let $\alpha$ be adaptable, and enable the self-adaptive component to adapt its value based on the network status, which can be analyzed according to the occurrences of connectivity failures at runtime.

Variable $\text{temperature}$ denotes the degree of freedom to select among the candidates. As we known, even a highly available service may fail occasionally due to network failures or other problems; meanwhile, at the beginning, it is difficult for the component to differentiate the candidates based on insufficient historical invocation feedback. Therefore, a high $\text{temperature}$ is given in the initial learning stage, thus
the candidates will have similar estimated values (i.e., similar probability to be selected). As the component gains more experiences, the temperature is decreased continuously and the estimated value becomes more sensitive to the value of history. When the temperature drops to a minimum, the probability of the most available service candidate will be significantly different from those of the less available ones.

During the annealing process, temperature is adjusted as follows:

**Equation (2):** Formally, let ‘$T$’ be the temperature and $minT$ be its lower bound. Let ‘$minCT$’ and ‘$totalCT$’ denote the minimum and the sum of the historical invocation times of service candidates, respectively. Let $\Delta$ be the incremental operator. Then,

$$\Delta T_{\text{anneal}} = -\min\{T - minT, \Delta \lfloor \frac{\text{totalCT}}{i} \rfloor + \Delta \text{minCT}\}$$  \hspace{1cm} (2)

where ‘$i$’ is the adjusting parameter. In our experiment, its value is set as 50, and ‘$minT$’ is set as 1.

To decrease the probability of being trapped in a local minimum, the self-adaptive component should be encouraged to try different service candidates, which means the temperature should be high as aforementioned. On the other hand, when the component has gained enough invocation feedback, the annealing process should be speeded up to ensure the algorithm’s efficiency. Consequently, the critical issue is how to measure the “mature degree” of the self-adaptive component. In Eq. (2) we measure it by considering two parameters. One is “the total amount of invocations in the history” ($\text{totalCT}$), whose impact is adjusted by a constant $i$. That is, the increase of the total invocation times means the component gets more experience. However, only the $\text{totalCT}$ is not enough, because it is possible that most of the invocations are sent to a small part of the candidates so in fact the self-adaptive component has little knowledge about the rest of the services. Thus, we use another parameter to record the service candidate with the minimal invocation times ($\text{minCT}$). The increase of this variable means the component gains more experience about the most unfamiliar candidate. The annealing process ends when temperature drops to $minT$.

On the other hand, a service candidate with a high success rate in previous invocations may become unavailable at runtime; meanwhile, the component may add new service candidates dynamically. In these cases, we should increase the temperature to provide other candidates (or the new services) with more opportunities. And it is called the ‘heating process’ (Eq. (3)).

**Equation (3):** Formally, for a service candidate, let ‘$recentSR$’ and ‘$hisSR$’ denote the success rates of the recent ‘$j$’ times of invocations and the historical invocations, respectively. During the heating process, $T$ is adjusted as follows:

$$\Delta T_{\text{heat}} = \begin{cases} m & \text{(if recentSR} - \text{hisSR} > 0.1 \text{ and } T = minT); \\ n & \text{(if a new service is added)}; \\ 0 & \text{(other cases)}. \end{cases}$$  \hspace{1cm} (3)
in this formula, ‘m’ and ‘n’ are the heating parameters. In our experiment, ‘j’ is set as 40, and the values of ‘m’ and ‘n’ are set as 20 and 50, respectively.

This formula represents that if the difference between recentSR and hisSR is more than 10% and the current temperature has been decreased to the lower bound, the temperature should be increased by m; note that after the heating process, the changed candidate’s old invocation records will be removed, because its recent availability seems much different from the historical records and should be re-evaluated. If a new service is added into the candidate list, we will increase the temperature by n.

After an invocation, the component will receive a feedback that contains the invoked candidate and the invocation result (success or failure). Then the component will adjust its estimated value of candidate availability as follows:

Algorithm 1. Adjust the candidates’ estimated values when a new feedback is received.
1. If a new feedback is received, load the required service’s candidate list from dependency table.
2. Add this feedback into the list of history records. Update the values of totalCT, minCT, recentSR, hisSR, etc.
3. Calculate $\Delta T_{\text{anneal}}$ and $\Delta T_{\text{heat}}$ based on Eqs. (2), (3). Update the value of temperature.
4. Calculate the candidates’ estimated values of availability based on Eq. (1).
5. Normalize these values so that the sum will keep being 1 to facilitate the selection; that is, $v_i = \frac{v_i}{\sum v_i}$.

A self-adaptive component can add a new service provider to its candidate list at runtime, and the candidates’ values will be adjusted as follows:

Algorithm 2. Adjust existing candidates’ estimated values when a new provider is added.
1. Load the required service’s candidate list from dependency table and add the new service provider.
2. Reset $\text{minCT}$ to 1.
3. Calculate $\Delta T_{\text{heat}}$ based on Eq. (3). Update the value of temperature.
4. Recalculate the candidates’ estimated values of availability based on Eq. (1).

As the historical records become rich and the temperature is gradually decreased, the component will have the highest probability to select the most available candidate. We hypothesize the correctness of our algorithm according to the law of large numbers [11] in statistics. We will report on the proof in the future.
4. Experimental Study

Our experimental study is divided into two stages. First, we evaluate our learning algorithm by comparing it with some general algorithms; in the experiment, we use a component with several service candidates that may change their availability dynamically. Then, we build a system with multiple self-adaptive components to observe the overall availability of the system. Because we cannot change the availability of the public services used in practical systems, for the requirements of experiments, we programmed several simulated services, whose availability can be changed by us at runtime.

4.1. Evaluation of the learning algorithm

Figure 5 shows the self-adaptive component and the services used in our first experiment. For simplicity, we assume the self-adaptive component’s computation is always available (i.e., the availability is 100%), whereas the simulated services may throw an exception according to a specified probability.

In this experiment, a client program will generate service requests to the component continuously at a fixed speed (about 200 requests/per minute). The component will select one of the function-identical services to delegate a part of its tasks; after eight minutes, we change the Service1’s availability to 60% to simulate a network problem (stage 2); and after another eight minutes, we add two different new services to the component’s candidate list (stage 3).

Two general algorithms are also implemented to compare with our learning algorithm (named as LA). The algorithm named test-and-choose (T&C) is to select every service candidate n times first, and then choose the service with highest availability. Another algorithm named recent-successful-choice (RSC) is to select a service randomly, and then choose the same one at the next time if this invocation is successful; otherwise (the invocation is failed), the component delegates the following request to another service candidate randomly.

Figure 6 shows the experimental results obtained, which have been averaged over 50 runs.

![Fig. 5. The experiment to evaluate the learning algorithm.](image-url)
As shown in Fig. 6, at stage 1, the component using the learning algorithm finds the high quality candidate fast and achieves highest availability; the T&C algorithm (the line tagged with squares) may find the best candidate (the success rate depends on the test times $n$, which is set as 50 in our experiments), but it has to spend a period of time in invoking each candidate many times to make decision. At stage 2, although both LA (the line tagged with diamonds) and RSC (the line tagged with triangles) can switch to another candidate quickly, the LA algorithm can find the new highest available candidate (i.e. Service3) and send most requests to it, thus gains a better availability. On the other hand, the T&C algorithm cannot detect the change of service availability, which may occur frequently in a dynamic environment. Experimental results also show that although RSC algorithm has a quick response to invocation failures by sending the following requests to another candidate, this strategy cannot recognize the best candidate so that the system cannot obtain the theoretically maximal availability after the environment becomes stable later. Therefore, we believe that the LA algorithm is effective; based on our approach, systems can gain high availabilities under the dynamic and decentralized environments.

4.2. Evaluation of self-adaptive component-based systems

Experiment 2 is to test the overall availability of a self-adaptive component-based system. In Fig. 7, there are three self-adaptive components and six services in the test environment. As mentioned in Sec. 2.3, services need to process requests from different clients, and every request will have an influence on the self-adaptive
components’ decision-making. To simulate the real environment, for each self-adaptive component, we place a client to send requests simultaneously. The component A is our test’s target.

At the early stage, components B and C have not found their best dependent service candidates; as a result, their own availabilities are not stable, and from the perspective of component A, the availabilities of components B and C keep changing at runtime. As components B and C gain more experiences from invoking their candidates to finish the tasks delegated from component A and other clients, these self-adaptive components will recognize their best candidates based on the learning algorithm and send most of the requests to them. Consequently, their availabilities reach their maximal values and become stable. Meantime, component A will be able to find its most available candidate. Figure 8 shows the experimental result, which represents that the system finds the best invocation process quickly (note that the path labeled by ‘h’ in Fig. 7 is the only path whose overall availability exceeds 80%), and with the run times increasing, the availability of the system becomes higher and gradually tends to the theoretically maximal value.
5. Implementation

The development and running of a self-adaptive component-based system can be described as follows: first, the self-adaptive components are developed and deployed independently; second, for the open decentralized systems, one or more registry services are initialized, they are waiting for self-adaptive components and services to join the system, and then inject into the joined self-adaptive components with their subscribed services; finally, these self-adaptive components become valid, behaves autonomously with the support of platform services.

To increase the feasibility of our approach, we implement our component model on top of the Spring framework [19]. The Spring framework provides a complete Java/JEE middleware service stack as well as an extensible framework, and thus we can reuse all the general-purpose services required by the components (such as transaction, security, interoperability) and focus on developing the value-added self-adaptation services. Moreover, because of the aspect-oriented techniques [20] adopted in the Spring framework, most of our implemented modules do not depend on the platform’s APIs, thus the concepts embodied in our model and the related implementations can be applied to other component platforms conveniently.

List 1 gives a portion of the specification for a self-adaptive component implemented in a sample application. The component specification adopts a spring-compliant style. In this application, the online media player component provides

```xml
<bean id="StreamingMediaPlayer"
    class="edu.pku.sei.component.SelfAdaptiveComponent">
    // component name
    // Provided Interface
    <property name="RequiredInterface" auto="false">
        <value> com.test.common.MusicBase </value>
    </property>
    // Dependency
    <property name="Dependency">
        <list>
            <ref bean="SOAPMusicService1" />
            <ref bean="RestfulMusicService2" />
        </list>
    </property>
</bean>

<bean id="RestfulMusicService2"
    class="edu.pku.sei.service.RestfulServiceStub">
    // Provided Interface
    <property name="params"> ... </property>
</bean>
```

List 1 a portion of the specification for a self-adaptive component
streaming media service for mobile devices. This component depends on a media service, from which it searches for the required media resource and caches it.

Similarly with traditional component model, developers can configure component dependencies explicitly in the specification (in this example, the SOAPMusicService1 service and the RestfulMusicService2 service), and these services will be loaded automatically by the component for further selection. However, for a self-adaptive component situated in the open and dynamic environment, all of its depended services may autonomously and dynamically enter or quit the environment. It is unpractical to statically list and bind all of the depended computations in the component specification. Therefore, in the specification of a self-adaptive component, developers can only specify the required service interface and set the attribute “auto” to “true”. In this mode, when the self-adaptive component is initialized, it will first access the registry service to subscribe the required interface as introduced in Sec. 2.3.

In the current implementation, we develop a specific component container, named SA (Self-Adaptive) container, to support the run of self-adaptive components (Fig. 9). The SA container provides the running space for a self-adaptive component instance, managing the life cycle of the component instance and the communications between the instance and others. The Dependency Manager encapsulates the self-adaptation related modules shown in Fig. 1; it maintains the service candidates required by the computation logic, and returns the most available one when the component instance needs to invoke the service. Based on the Spring-AOP mechanism [19], the invocation results are intercepted by a monitor and sent to the Dependency Manager to update its knowledge. Similarly, a set of system services (transaction, security, etc.) are woven into the component logic as aspects to intercept and process the invocations.
An Instance Manager is responsible for the instance creation of a kind of self-adaptive component. Although the experiments have shown that the proposed algorithm is effective to find the most available service provider from many third-party candidates, as situated in a dynamic environment and with unknown service providers, a self-adaptive component has to take a period of time to learn from the feedback of invocations. To further reduce the learning cost, one way is to “learn from others”, that is, to enrich a component instance’s experience by reusing other instances’ invocation results. In our current implementation, the Instance Manager enables the component instances to share knowledge among each other. As shown in Fig. 10, for each self-adaptive component type, an Instance Manager will record all the active component instances and their invocation times, and when a new component instance is created, it will be initialized with the most “experienced” instance’s invocation history. That is, this new instance will perform as good as the trained instances without a learning curve in its early stage.

6. Related Work

Many research efforts have focused on how to improve the system availability at runtime. In some distributed systems (e.g., systems built on the internal services within an enterprise boundary), system components are deployed throughout a group of machines, and the lifecycles of components are controlled by a centralized management server. The availability of this kind of systems can be improved via system redeployment [12]. The redeployment process mainly includes the following steps: first, the management server monitors and calculates the reliability of the underlying network; then, the server determines the optimal deployment plan based on the components’ availability, their interaction frequencies, the reliability of network connectivity, etc; finally, the server redeploys the system via component migration. However, in decentralized systems, redeployment techniques are unsuitable because components deployed in third-party provider’s machine cannot be migrated. Moreover, a service’s availability may change at runtime, whereas existing redeployment algorithms often consider it as a static value.

Availability is an important aspect of Quality of Service (QoS). Several related research projects have adopted broker-based solutions to address this problem.
The broker architecture maintains the services’ functionality and quality information on a specific component (the broker); when a service requestor asks for a service, the broker finds the most available service provider based on a selection algorithm, forwards the service request to the chosen provider, and then sends the result back to the requestor. For example, Zeng et al. [13] presents a platform which addresses the issue of selecting web services for the purpose of their composition in a way that maximizes user satisfaction expressed as utility functions over QoS attributes (availability, response time, etc).

To some extent, a self-adaptive component acts like a broker since it stores the information about multiple functional-equivalence service providers for choosing one of them for services. However, a self-adaptive component is also different from a broker in many aspects. From the functionality perspective, the main difference is that a broker is not a part of the system’s business logic, while a self-adaptive component is implemented to perform business functions. From the non-functionality perspective, because the proposed algorithm supports the self-adaptive components to find the most available service candidates themselves instead of delegating to a remote broker (no matter whether multiple broker instances exist or not), the proposed approach is more effective than the broker-based solutions since both network latency and maintenance complexity are reduced by removing the intermediate layer.

Service brokers often use some service selection algorithms to achieve the system goals (e.g. load-balancing, performance optimization). To the best of our knowledge, most of the existing algorithms are not designed for the decentralized systems discussed in our paper: Comparing with these algorithms, our proposed solution enables self-adaptive components to estimate the availability of remote services via the feedback and to make decisions without requiring the services to provide their running information. This feature is important in the decentralized systems, because most of the services are independent and autonomous, and the availability information is often not published to the clients.

Several existing research projects use machine learning techniques to adapt systems to the changing environments. Doshi et al. [14] use Markov decision process (MDP) to model web service workflow composition. They interleave MDP-based workflow generation and Bayesian model learning to produce robust workflows. In that approach, the optional services are predefined and cannot be modified at runtime. Dowling [15] proposed an adaptive component framework named K-Component, which facilitates building autonomic systems by collaborative reinforcement learning. In this model, adaptation actions of the adaptive components are fixed after the compile stage, thus how to deal with new coming services is not addressed.

7. Conclusion and Future Work

Nowadays, more and more software systems are constructed with third-party services, and deployed in a decentralized environment like the Internet. Because of the
instability of network connections and the autonomy of independent services, how to achieve high system availability becomes a crucial challenge in the system construction. Most of the existing approaches employ an external central component to cluster many function-identical services and then find the best one. However, as mentioned in the previous sections, these centralized approaches may suffer from the single-point failure and are not suitable for a client-dependent quality property like availability.

Therefore, in this paper, we proposed an approach to constructing high-available decentralized systems based on our self-adaptive component model. In this model, self-adaptive components are initialized with their depended services based on a service discovery mechanism adopted in the hierarchical P2P systems. To adapt to the dynamic environment, self-adaptive components dynamically estimate the availabilities of remote entities at runtime, and update their knowledge via learning from the feedback of historical invocations. Based on the estimation, the self-adaptive components select interaction entities autonomously to self-organize a high available runtime system. We validated the effectiveness of the learning algorithm through several experiments, and implemented the runtime framework and assisting tools on top of mature platforms.

In our current implementation, the functional-identical services need to implement the same interface, i.e., the same method signatures (method name, parameter name and order). But in practice, we find that components often do not have the choice between several alternative services because they have minor differences in their interface. Currently we are working on eliminating the mismatches semi-automatically based on a heuristic approach.

For the future, since whether the proposed algorithm is also effective in large-scale practical systems is unclear, we are working on evaluating it in practical real-time systems. We also plan to investigate further the algorithm to concern multiple system quality attributes (such as security, response time) for the purpose of building high-quality decentralized systems.

Acknowledgements
The authors would like to thank the anonymous reviewers for their helpful comments on the earlier draft of this paper. We would also like to thank Yan Li, and other members of the autonomous computing group from the Institute of Software, Peking University. The research was partially sponsored by the National Grand Fundamental Research 973 Program of China under Grant No. 2005CB321805 and the National Nature Science Foundation of China under Grant No. 90412011, 60773151.

References