Building Self-adapting Services Using Service-specific Knowledge

An-Cheng Huang[†] and Peter Steenkiste[†‡]
Computer Science Department[†] Electrical and Computer Engineering Department[‡]
Carnegie Mellon University
{pach,prs}@cs.cmu.edu

Abstract

With the advances in middleware and Web services technologies, network services are evolving from simple client-server applications to self-configuring services that can compose primitive components distributed in the Internet into a value-added service configuration that provides rich functionalities to users. A resulting research problem is how to continuously adapt such composite service configurations at run time in order to cope with the increasingly dynamic and heterogeneous network environments and computing platforms. In this paper, we propose a self-adaptation architecture that allows service developers to specify their service-specific adaptation knowledge as "externalized" adaptation strategies. These adaptation strategies are used by a general, shared adaptation framework to perform run-time adaptation operations that automatically incorporate service-specific knowledge. In addition to the strategies, we also identify another aspect of adaptation knowledge that is not addressed by previous solutions: adaptation coordination. Our framework provides integrated support for the specification and execution of both aspects of developers' adaptation knowledge.

1. Introduction

Network applications such as Web browsing, video conferencing, instant messaging, file sharing, and online gaming are becoming a necessity for more and more people. From a user's perspective, these network applications are used to access *services* offered by *service developers* over the Internet. Advances in middleware and Web services technologies have enabled service developers to build value-added services using distributed software components to satisfy particular user requirements. In order to maintain the service performance and quality, such distributed composite services must be able to dynamically adapt their configurations to the frequent run-time changes in network characteristics (e.g., latency and bandwidth), resource availability (e.g., CPU and memory), and other environment factors.

In many cases, how to perform run-time adaptation is highly service-specific, i.e., simply using some generic adaptation heuristics is not sufficient, and *service-specific knowledge* is required to appropriately adapt the service configuration at run time. Many previous research efforts to support run-time adaptation adopt an "internalized" approach that requires developers to integrate their service-specific *adaptation strategies* into the target system, e.g., [16, 19, 9, 3]. While this approach is flexible, it forces a developer to hard-wire the knowledge into the system, increasing design complexity and development cost.

In addition to the adaptation strategies, adaptation coordination is another important aspect of distributed composite services that requires service-specific knowledge. Multiple strategies may be invoked at the same time, and they may want to make conflicting changes to the configuration. Furthermore, the developer may design strategies that "at cross purposes", e.g., one strategy adds a server to improve performance while another strategy removes a server to reduce cost. Such adaptation coordination issues are a challenging problem that is not addressed by previous solutions.

In this paper, we present a self-adaptation architecture that allows service developers to easily add run-time adaptation capability to their services. We use an "externalized" approach adopted in several previous studies (e.g., [11, 26]): we define a representation for developers to express their service-specific adaptation knowledge in the form of externalized strategies and coordination policies, and we build a general framework that can interpret such knowledge to automatically adapt the target system at run time. Since the general, shared framework provides common adaptation functionalities, our approach reduces the development cost as the developers do not need to worry about lower-level mechanisms.

In the rest of the paper we define the run-time adaptation

This research was funded in part by NSF under award number CCR-0205266.

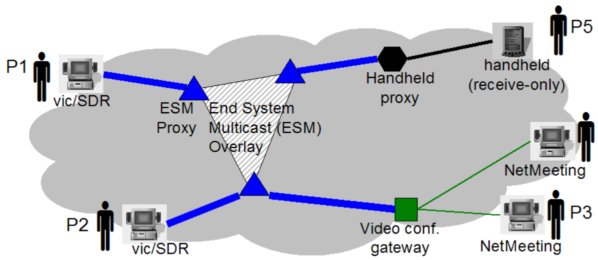

Figure 1. A video conferencing service.

problem, present the self-adaptation architecture, discuss the support for service-specific adaptation strategies and coordination policies, describe our prototype implementation, and use simulation to demonstrate the advantages of our approach.

2. Problem statement

We focus on the problem of adding run-time adaptation capabilities to a composite service. Therefore, we assume that the initial configuration of a service is already constructed. One possibility is that the developer constructs the configuration manually. Alternatively, the developer can build a "self-configuring service" using an existing self-configuration framework, e.g., [13, 12, 22, 1], that automatically composes the service configuration.

Given the initial service configuration, we have identified two aspects of a developer's service-specific knowledge that can be used to guide the adaptation of the configuration at run time: adaptation strategies and coordination policies. Let us first use an example to illustrate these. In Figure 1, five users want to hold a video conference. Two use MBone conferencing applications vic/SDR (VIC), two use NetMeeting (NM), and one uses a receive-only handheld device (HH). Suppose the initial configuration for the users consists of a video conferencing gateway (VGW) that translates different conferencing protocols, a handheld proxy (HHP) joining the session on behalf of HH, and three End System Multicast (ESM) proxies that provide wide-area multicast functionality for the users.

At run time, the service configuration needs to be adapted to accommodate environmental changes. For example, to handle run-time problems such as high load and congestion at the VGW, the developer may have the following two strategies:

- S1: (VGW overloaded) → (replace VGW with a high-capacity VGW)
- S2: (VGW congestion) → (replace VGW with a high-bandwidth VGW)

These strategies are service-specific because another developer may have different strategies, e.g., VGW overload and congestion could be handled by reducing the codec quality and bit rate.

Another important aspect of the adaptation knowledge is how the strategies should be "coordinated". For example, suppose at run time, S1 and S2 above are invoked at the same time. If the two strategies want to replace the VGW with different candidates, obviously only one of them should be allowed to execute.

Our goal is to build an adaptation framework that allows a developer to add self-adaptation capabilities to a service and adapts the system based on the developer's knowledge. In this paper, we limit our scope to *local adaptation*, i.e., we focus on how to support adaptation strategies involving only "local" actions and how to coordinate such strategies. Specifically, a local adaptation only changes a single component (e.g., changing the parameters of a component, replacing a component, etc.) and has limited "indirect" effects (e.g., it only affects the component it changes).

3. Overview of adaptation architecture

In our architecture (Figure 2), a developer uses our knowledge representation to express its service-specific adaptation knowledge, including *adaptation strategies* and *coordination policies*. This knowledge, along with the *initial configuration* of the service, are given to the self-adaptation framework, which consists of an *adaptation manager* (AM), an *adaptation coordinator* (AC), and the supporting infrastructure.

The supporting infrastructure provides the common functionality required for run-time adaptation, e.g., a network measurement infrastructure for measuring critical network performance metrics, a service discovery infrastructure for finding new components, a component management/deployment infrastructure for controlling/deploying the components, etc. In this paper, we assume the necessary infrastructures exist, and we focus on how the developers' knowledge can be represented and on how the AM and the AC can make use of the adaptation and coordination knowledge to adapt the target service configuration.

At run time, the AM handles a developer's adaptation strategies by monitoring the configuration to detect run-time problems. When a problem occurs, e.g., a component becomes overloaded, and one of the developer's strategies is designed to handle the problem, the AM invokes the strategy to adapt the configuration, e.g., replace the overloaded component. When a strategy is executed, it generates a *proposal* specifying how it wants to change the configuration, and the proposal is sent to the AC. If a proposal does not conflict with other proposals, the AC accepts the proposal and asks the AM to change the configuration accordingly.

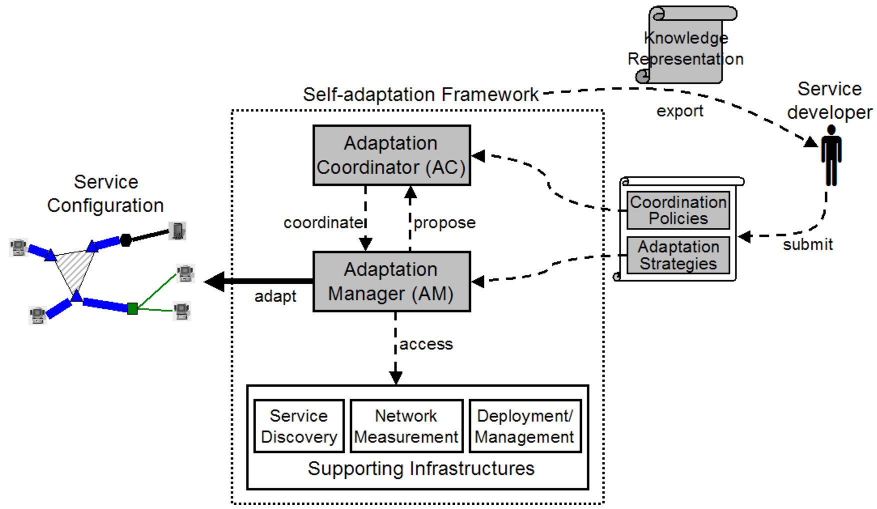

Figure 2. Architecture for run-time local adaptation support.

Otherwise, the AC rejects the proposal.

Next, we discuss our design of the knowledge representation for specifying the adaptation strategies and coordination policies, and we describe how such knowledge is used to perform run-time adaptation.

4. Adaptation strategies

Previous studies have proposed many run-time adaptation solutions based on "internalized" adaptation strategies (i.e., the adaptation logic and mechanisms are hard-wired into the system itself), for example, [16, 19, 9, 3]. While this approach gives the service developer complete control over how adaptations are performed, it typically results in high development costs.

The "externalized" approach adopted by some previous projects, e.g., [11, 26], addresses this problem by separating the strategies and mechanisms from the actual system. This enables the development of a general adaptation framework that can be reused by different systems, and the developer of a system can add run-time adaptation capabilities by designing externalized strategies without modifying system components. Therefore, the development cost is potentially much lower than with the internalized approach.

To support externalized strategies, one important design decision is how to specify such strategies. As categorized in [26], an adaptation strategy can be specified as a high-level "utility" function or an explicit "event-action" rule. The utility approach allows a service developer to specify a utility function indicating the "desirable" configurations, and the adaptation mechanisms automatically modify the service configuration towards higher utility. This approach is for example used in run-time adaptation solutions that focus on dynamic resource allocation, e.g., [17, 21, 26]. However, our scope of adaptation is much broader and includes component-level adaptations such as replacing/adding/removing components. Therefore, the utility approach is not feasible.

The event-action approach lets a developer specify rules dictating what "actions" should be taken when a particular "event" occurs. For example, when the event "component X becomes overloaded" occurs, the appropriate action is "replace X with a higher capacity one". This approach is used in previous studies where the target system involves different, heterogeneous components, e.g., [11, 8]. Therefore, we adopt the externalized, event-action approach for strategy specification.

4.1. Strategy format

Our support for adaptation strategies is built on the externalized approach presented in Rainbow [11]. An adaptation strategy consists of the following three parts.

- *Constraint*: A strategy is invoked when its constraint is "violated". A constraint is a condition on certain properties of the configuration, e.g., "load(X) $< C$" where X is a component in the configuration. The properties can be performance metrics, e.g., bandwidth and latency, or component properties. At run time, the AM monitors the constraints of the strategies and invokes a strategy when its condition becomes false.

- *Problem determination*: A constraint violation may be caused by multiple *triggering problems*. When a strategy is invoked, it may need to, for example, query more specific configuration properties to determine the actual triggering problem. For example, in the video conferencing example, a strategy triggered by "low HH video quality" needs to determine whether the actual problem is HHP failure, low quality codec used by the HHP, or congestion at the HHP.

- *Tactic*: A tactic consists of a set of *actions* that are used to address a particular triggering problem. Actions range from changing a run-time parameter of a component to changing the configuration by inserting/removing components. In the example above, "HHP failure" can be addressed by a tactic that replaces the failed HHP with a new one, "low quality codec" can be addressed by "increasing the codec quality", and so on.

4.2. Strategy specification

We now discuss how each part of a strategy is specified. Our self-adaptation framework is based on the self-configuration framework we built previously [13]. Specifically, we leverage the existing *abstract configuration API* and *objective function API*. The abstract configuration API includes data structures representing components and component types in a service configuration and functions for adding/removing/connecting components in the configuration. We assume the initial configuration is given to our self-adaptation framework as a data structure that is constructed using this API. Therefore, an adaptation strategy designed by a developer can "reference" components or component types in the configuration. The objective function API includes data structures representing performance metrics and other properties of components/connections in the configuration and operators in an objective function for component selection.

In addition to the above existing functionality, new data structures and functions are needed for specifying all parts of a strategy. Table 1 summarizes the additions. We now describe how a developer can use these APIs to specify each part of an adaptation strategy.

- *Constraint*: A developer can use the objective function API to construct a function of the relevant properties of components and connections in the configuration. A constraint can then be constructed using `RelationOp` and `BooleanOp` with the function. When a constraint is violated, the "violator" in the configuration is passed to the corresponding strategy (similar to Rainbow [11]), which can then operate on the appropriate component.

- *Problem determination*: Since our framework is implemented in Java and exports Java interfaces and classes for specifying strategies, a strategy designed by a developer is basically a small Java class. This approach gives the developer significant flexibility in implementing the problem determination logic.

Data structure
`RelationOp`: Represent relation operators such as "==", ">=", etc.
`BooleanOp`: Represent boolean operators such as "AND", "OR", etc.
`Condition`: Represent a combination of "`Term RelationOp Term`".
`Constraint`: Represent a combination of "`Condition BooleanOp Condition`".
`Tactic`: Represent the base class for a tactic; developers implement tactics by creating specializations.
`Strategy`: Represent the base class for a strategy; developers implement strategies by creating specializations.

Function
`replaceComponent(c)`: Represent an adaptation action that replaces an existing component `c` in the current configuration.
`changeParameter(pn,pv)`: Represent an adaptation action that changes the value of the parameter `pn` of a component to `pv`.
`connect(c1,c2)`: Represent an adaptation action that connects components `c1` and `c2`.
`setTacticObjective(obj)`: Set the component selection objective for a tactic to `obj`, which will be used for, e.g., the `replaceComponent` actions.
`setConstraint(C)`: Associate the constraint `C` with an adaptation strategy.
`invokeTactic(T)`: Invoke the tactic `T`.

Table 1. API for specifying adaptation strategies.

- *Tactic*: Since we focus on "local adaptation", our API allows a tactic to specify actions such as `replaceComponent`, `changeParameter`, and `connect`. Similar to our previous self-configuration framework, when a tactic requires a new component, it specifies an objective function as the component selection criterion using `setTacticObjective`. The support infrastructure will then use this objective function to select the best server to execute the component, given current runtime conditions.

- *Strategy*: Finally, we need a data structure to represent a strategy. The constraint of a strategy can be assigned using `setConstraint`, and a strategy can invoke a

`tacticNewNM`: This tactic connects a new NM user to the VGW.
`tacticNewVIC`: Connect a new VIC user to the closest ESMP.
`tacticNewHH`: Connect a new HH user to the HHP.
`tacticVGWFail`: Replace a failed VGW with a high-capacity one.
`tacticVGWOverload`: Replace an overloaded VGW with a high-capacity one.
`tacticVGWCongest`: Replace a congested VGW with a high-bandwidth one.
`tacticVGWLowQual`: Increase the VGW's codec quality.
`tacticESMPFail`: Replace a failed ESMP with a high-bandwidth one.
`tacticESMPCongest`: Replace a congested ESMP with a high-bandwidth one.

Table 2. Tactics for the video conferencing service.

particular tactic using `invokeTactic`.

We believe our self-adaptation framework can also be applied to other component-based services frameworks such as Ninja [12], SWORD [22], and ACE [1]. The major requirement is that such a framework (1) provides a representation of the service configuration allowing our framework to reference the components in the configuration and (2) provides an interface allowing our framework to make changes to the configuration according to the developers' knowledge.

4.3. Example

We use the video conferencing service as an example to illustrate how a developer's adaptation strategies can be specified using the above APIs. Suppose the developer implements the tactics shown in Table 2. Based on these tactics, the developer then designs the strategies shown in Table 3 (for simplicity, the actual code is not shown). As seen in Table 3, the constraints for these strategies are as follows:

- C1: `(config.numUnconnectedUsers == 0)`
- C2: `(NM.videoQuality >= Tn)`
- C3: `(VIC.videoQuality >= Tv)`

Therefore, the adaptation strategies are instantiated and associated with their constraints using the following statements:

`stratNewUser`: This strategy is triggered when a new user joins the session. It determines the triggering problem and invokes the appropriate tactic as follows. New NM user → `tacticNewNM` New VIC user → `tacticNewVIC` New HH user → `tacticNewHH`
`stratNMQual`: Triggered when the video quality of NM is below a threshold T_n. VGW failed → `tacticVGWFail` VGW overloaded → `tacticVGWOverload` VGW congested → `tacticVGWCongest` VGW poor codec → `tacticVGWLowQual`
`stratVICQual`: Triggered when the video quality of VIC is below a threshold T_v. ESMP failed → `tacticESMPFail` ESMP congested → `tacticESMPCongest`

Table 3. Strategies for the video conferencing service.

```
stratNewUser S1 = new stratNewUser();
S1.setConstraint(C1);
stratNMQual S2 = new stratNMQual();
S2.setConstraint(C2);
stratVICQual S3 = new stratVICQual();
S3.setConstraint(C3);
```

The strategies are given to the AM, which monitors the configuration and invokes a strategy when its constraint is violated.

5. Adaptation coordination

Our goal with respect to adaptation coordination is to only require a service developer to specify the service-specific coordination knowledge without worrying about the underlying mechanisms. We identified three important coordination issues: detecting conflicts between proposals, resolving conflicts between proposals, and identifying incompatible strategies (i.e., strategies that work at cross purposes). Next, we discuss how we address these three issues.

5.1. Conflict detection

We categorize conflicts into two types: *action-level* and *problem-level*. An action-level conflict occurs when two proposals want to make "conflicting changes" to the configuration. For example, if one proposal wants to replace server A with B, and another proposal wants to replace A with C, then obviously only one can be accepted. In other words, the two proposals attempt to change the same "target" in different ways. The AC can automatically detect such conflicts by looking at the actions in different proposals.

A problem-level conflict occurs when the "intentions" of two proposals conflict with each other, i.e., they are addressing two problems that should not be addressed at the same time. For example, strategy S1 connects a new VIC user to the closest ESMP in the configuration, and S2 replaces a failed ESMP with a new one. Suppose there are three ESMPs (A, B, and C) in the configuration. User U wants to join the video conference, and at the same time C fails; as a result, both S1 and S2 are invoked. Among A, B, and C, B is closest to U, so S1 proposes to connect U to B. At the same time, S2 proposes to replace C with D. Since D is closer to U than B is, a developer may want to delay S1 until after C has been replaced with D. In other words, there is a problem-level conflict between the proposals of S1 and S2 (i.e., the "new VIC user" and "ESMP failure" triggering problems should not be addressed at the same time).

However, this is not the only solution. Another developer may prefer S1 and S2 to be executed together so that the new user join will not be delayed. In other words, they do not consider this a problem-level conflict. Therefore, problem-level conflicts are service-specific and cannot be detected automatically. A developer must specify explicitly whether the triggering of two problems simultaneously constitutes a "problem-level conflict".

To allow a developer to specify problem-level conflicts, we observe that since each problem is addressed by a tactic, a problem-level conflict can be specified as a conflict between two tactics. Our framework provides the following function for specifying such a conflict between tactics T1 and T2.

```
addProblemConflict(T1, T2);
```

As an example, a developer for the video conferencing service may specify the following problem-level conflicts.

```
addProblemConflict(tacticNewNM,
                   tacticVGWFail);
addProblemConflict(tacticNewNM,
                   tacticVGWOverload);
addProblemConflict(tacticNewNM,
                   tacticVGWCongest);
addProblemConflict(tacticNewVIC,
                   tacticESMPFail);
addProblemConflict(tacticNewVIC,
                   tacticESMPCongest);
```

Based on this specification, the AC constructs a set of coordination policies and uses them to detect problem-level conflicts at run time.

As discussed earlier, when performing coordination we focus on the "direct" effects of an adaptation. Although it is difficult to detect "indirect" conflicts automatically, such conflicts can still be specified as problem-level conflicts if the developer knows that, for example, the actions of two tactics may conflict indirectly.

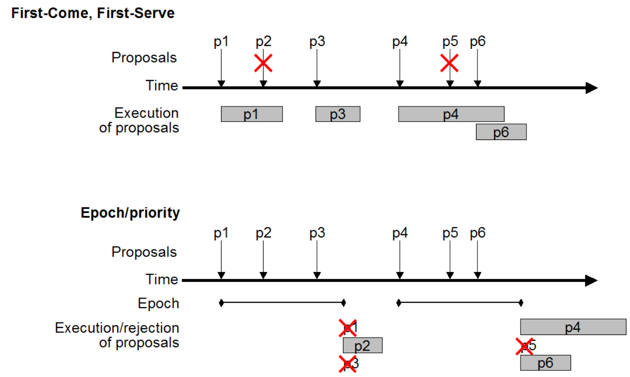

Figure 3. Two conflict resolution approaches.

5.2. Conflict resolution

Figure 3 illustrates two different approaches for resolving conflicts between proposals. There are six proposals, and conflicts exist between p1 and p2, between p2 and p3, and between p4 and p5. In addition, p2 has a higher priority than p1, i.e., p2 $>$ p1, and similarly, p2 $>$ p3 and p4 $>$ p5. We now briefly describe the two different approaches.

- *First-Come, First-Serve (FCFS)*: This approach accepts or rejects proposals as they are received. A proposal is accepted if no other conflicting proposals are being executed. If a proposal is received when another proposal is being executed, the AC performs conflict detection between the two proposals. If a conflict is detected, the new proposal is rejected.

  In the figure, p2 is rejected because p1 is proposed earlier and is being executed. Similarly, p5 is rejected since p4 is in progress. However, p6 is allowed to start since there is no conflict between p4 and p6.

- *Epoch/priority*: This approach divides time into discrete "epochs". At the end of an epoch, the AC performs conflict detection among all proposals received within the epoch. If proposals conflict, the one with the highest priority is accepted, and all the others are rejected. Priorities are assigned to tactics by the developer according to service-specific knowledge.

  In the figure, only p2 is allowed to execute in the first epoch. Similarly, p5 is rejected in the second epoch.

The FCFS approach supports more limited conflict resolution while the epoch/priority approach is more flexible.

However, the flexibility of the epoch/priority approach is gained by sacrificing "agility" [19]: all proposals within an epoch have to wait until the end of the epoch. For this reason, the epoch/priority approach is used for applications where simplifying assumptions about the timing of events can be made, e.g., coordinating rules in active databases [15, 5]. However, when coordinating adaptations in a distributed self-adapting service, a fast response to constraint violations is often important, e.g., to recover quickly from failures or poor performance. Therefore, we use the FCFS approach because of its agility.

When a proposal is rejected, the AC informs the AM of the decision. The AM can handle the rejection in different ways. If the triggering condition is still true, the AM can re-propose the same proposal immediately. However, if many strategies are triggered frequently, this may create contention at the AC, and therefore the AM may want to back off the retries. Another possibility is that the AM can drop the proposal, and the proposing strategy will be invoked again if the triggering condition is still true. In our evaluation, we use the simple mechanism that rejected proposals are retried immediately. However, understanding the effects of these different approaches requires further study.

5.3. Identifying incompatibility

To prevent adaptation strategies from working at cross purpose, we need to identify "incompatible" strategies. For example, let us assume that strategy S_a adds a new server when an existing server is overloaded, and S_r removes a server when existing servers are under-utilized. These two strategies are intended to maintain the system in an efficient operating region. However, if their triggering conditions are not defined carefully, they can potentially cause a "cycle" of adding/removing a server to/from the service, i.e., it can result in "thrashing".

If we want to automatically determine whether such cycles exist, The AC need to determine the exact effects of strategies (e.g., how much load is reduced by adding a server) and whether one strategy's effects will trigger another. Such analysis is difficult since it requires domain knowledge, and the exact run-time effects may be difficult to predict.

Instead of solving the general problem, we observe that it is usually sufficient to identify strategies that have opposite goals (and thus may cause undesirable cycles) and warn the developer. The developer can then verify that the goals are correct and that cycles will not occur. Note that although so far we have discussed incompatibility at the strategy level, we actually need to analyze incompatibility at the tactic-level since the unit of coordination in our architecture is tactics.

To automatically identify incompatible tactics, we let developers "annotate" the tactics with causes and effects. We assume that all causes and effects are changes in performance metrics, which can be specified using the objective function API. Therefore, our framework exports the following functions for cause and effect specification:

```
addTacticCauses(List increasedMetrics,
                List decreasedMetrics)
addTacticEffects(List increasedMetrics,
                 List decreasedMetrics)
```

Continuing the earlier example, suppose S_a invokes tactic `Ta`, and S_r invokes `Tr`. The tactics can then be annotated as follows:

```
// a1 { load }, a2 { }
// a3 { }, a4 { load }
Ta.addTacticCauses(a1, a2);
Ta.addTacticEffects(a3, a4);

// r1 { }, r2 { load }
// r3 { load }, r4 { }
Tr.addTacticCauses(r1, r2);
Tr.addTacticEffects(r3, r4);
```

This indicates, for example, that tactic `Ta` is invoked in response to an increase in load (i.e., the cause) and results in a reduction in load (i.e., the effect). Given this information, the potential cycle between the strategies can be automatically detected by the AC.

6. Implementation and evaluation

We have implemented a prototype of the self-adaptation framework based on our earlier self-configuration framework. As mentioned earlier, we added additional functionality to the knowledge specification APIs to allow developers to specify their strategies and coordination policies. We built the AM and the AC to handle the strategies and coordination at run time, respectively.

To evaluate our approach, we applied our framework to a simulated massively multiplayer online gaming service, depicted in Figure 4. Below we summarize the key simulation properties.

Service components. There are two types of nodes in a service configuration: users and servers. Users move around randomly in the virtual game space, and each server handles a partition of the space, including all the users within that partition.

Adaptation strategies. The gaming service has five adaptation strategies: *join* connects a new user to the corresponding server, *leave* disconnects a user from its server, *cross*

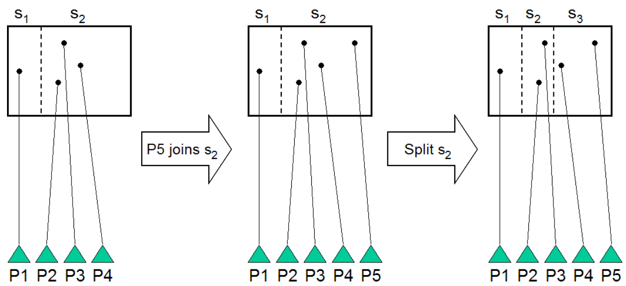

Figure 4. A massively multiplayer online gaming service.

moves a user's state from one server to another, *split* adds a new server when an existing one becomes overloaded, and *merge* removes a server by merging two under-utilized servers. Since each strategy only invokes one tactic, the names also refer to the tactics.

Simulator. We generate traces of user arrivals and departures where the inter-arrival time has an exponential distribution, and the stay duration has a bounded Pareto distribution. Each simulation has a duration of 500 minutes, and the average number of users at any time is 142.12. We implemented an event-driven simulator that takes such a trace and simulates the gaming service described above. The simulator is integrated with the self-adaptation framework, which interprets the above adaptation knowledge to perform run-time adaptation.

In this paper, we present two sets of simulation results to show that our approach allows developers to concentrate on the service-specific policies without worrying about the underlying mechanisms. First, let us assume that the gaming service developer uses our API to specify the following coordination policies.

```
addProblemConflict(join, split);
addProblemConflict(join, merge);
addProblemConflict(leave, split);
addProblemConflict(leave, merge);
addProblemConflict(cross, split);
addProblemConflict(cross, merge);
```

We assume that if a proposal is rejected, the proposing tactic will re-propose as soon as possible, i.e., a rejected adaptation is delayed. To see the effects of the coordination policies, Table 4 shows the number and percentage of delayed adaptations of each type. The percentage of delayed adaptations is much higher for split/merge operations than that for join/leave/cross operations. This is because there are way more join/leave/cross operations than split/merge operations. However, overall the impact of the delays is small

	Num. adapt.	Adapt. delayed	Percent. delayed
join	50149	258	0.51
leave	49992	384	0.77
cross	6194675	38146	0.62
split	789	715	90.62
merge	783	746	95.27

Table 4. Number and percentage of delayed adaptations

	Num. adapt.	Adapt. delayed	Percent. delayed
join	50149	184	0.37
leave	49992	0	0
cross	6375261	28528	0.45
split	623	573	91.97
merge	613	587	95.76

Table 5. Number and percentage of delayed adaptations: "no leave conflicts".

because (1) few join/leave/cross operations are delayed and (2) although most split/merge operations are delayed, they are much less frequent, and they are expensive anyway (requiring 520 ms without delays), so the delay (maximum about 100 ms) has limited impact.

To see how easily a developer can apply a different set of coordination policies, consider the following scenario. Suppose that the above developer improves the server implementation such that it is able to handle the departure of a user in parallel with other adaptations, i.e., the "leave" adaptation no longer conflicts with other adaptations. To take advantage of this new capability, the developer can simply remove the lines specifying conflicts that involve "leave", resulting in the following policies.

```
addProblemConflict(join, split);
addProblemConflict(join, merge);
addProblemConflict(cross, split);
addProblemConflict(cross, merge);
```

We perform another simulation (with the same parameters except the policies) to verify that such a simple change in the specification indeed results in the expected coordination behavior at run time. Table 5 shows the number and percentage of delayed adaptations of each type.

Of course, as expected, no leave adaptations are delayed with the new policies. Furthermore, eliminating "leave conflicts" actually results in fewer split/merge adaptations. This is because having fewer leave conflicts stabilizes the configuration such that fewer split/merge operations are necessary. In turn, this allows more join/cross adaptations to be executed without delay. Finally, because the users now spend less time "waiting" (i.e., being delayed), they have more time to move around in the game space, resulting in more cross adaptations as seen in Table 5.

This example demonstrates that our approach of separating the knowledge from the mechanisms allows a service developer to easily implement service-specific coordination policies without worrying about the underlying coordination mechanisms.

7. Related work

Many projects have studied ways of adding run-time adaptation capabilities to different types of systems. For example, some projects have focused on communication adaptation in a client-server system, e.g., [16, 19, 6, 2]. Other studies use similar "parameter-level" adaptation techniques in more general distributed systems, e.g., [9, 21, 3, 8]. Another class of adaptation solutions is based on dynamic resource allocation using utility functions, e.g., [7, 26, 17], or using application models specifying resource requirements, e.g., the ARA mechanisms [24] in the RT-ARM system [14]. In the context of high performance computing, adaptation mechanisms have also been developed to cope with changes in the run-time environment, e.g., the QuO framework monitors and adapts applications according to their QoS contracts [18]; the adaptation framework in [4] determines when to adapt, and it uses a tunability interface to modify the run-time parameters of applications. Since we target both component-level and parameter-level adaptations in distributed composite services, previous component-level adaptation solutions such as [23, 10, 25, 20, 11] are more relevant to our work.

Many of the above solutions rely on generic adaptation heuristics. As discussed throughout this paper, such heuristics may not be sufficient in many cases, and service-specific knowledge may be required. Of course, for a large-scale system, specifying all the necessary knowledge may be a tedious task for the developer. Therefore, for generic adaptation scenarios that do not require service-specific knowledge, the mechanisms developed in previous generic solutions may be leveraged to make our framework more "intelligent".

Our work is built on the externalized event-action approach for specifying adaptation strategies, similar to Rainbow [11]. However, one difference is that Rainbow supports more global adaptation strategies while we focus on local adaptation. Secondly, while Rainbow enables developers to choose the most appropriate architectural style for adaptation, we leave it to developers to design service-specific strategies. Finally, Rainbow and other previous solutions do not address coordination issues. In contrast, we provide integrated support for conflict detection, conflict resolution, and identification of incompatible strategies.

Previous run-time adaptation solutions do not explicitly support adaptation coordination. Some use a single monolithic adaptation strategy that makes all adaptation decisions, so conflicts cannot occur. Others divide the target system into partitions and assume adaptations from different parts are independent. Most related to our work are studies that look at coordinating the execution of event-condition-action policies [5] and coordinating update rules in active database systems [15]. However, they adopt the epoch/priority approach for conflict detection and resolution and therefore rely on assumptions that do not hold in our context.

8. Conclusions

We presented a reusable self-adaptation framework that provides common adaptation functionality and yet can take advantage of developers' service-specific adaptation knowledge. Our framework allows a developer to specify not only adaptation strategies but also coordination policies. We identified and addressed three adaptation coordination issues: conflict detection, conflict resolution, and identifying incompatible strategies. We implemented a prototype of the framework and evaluated our solution using a simulated gaming service. Results show that coordination works as expected and that our approach also allows developers to easily design and change coordination policies.

References

[1] M. Agarwal and M. Parashar. Enabling Autonomic Compositions in Grid Environments. In *Proceedings of the Fourth International Workshop on Grid Computing (Grid 2003)*, pages 34–41, Nov. 2003.

[2] R. K. Balan, M. Satyanarayanan, S. Park, and T. Okoshi. Tactics-Based Remote Execution for Mobile Computing. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys '03)*, pages 273–286, May 2003.

[3] J. W. Cangussu, K. Cooper, and C. Li. A Control Theory Based Framework for Dynamic Adaptable Systems. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC '04)*, pages 1546–1553, Mar. 2004.

[4] F. Chang and V. Karamcheti. Automatic Configuration and Run-time Adaptation of Distributed Applications. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC-9)*, pages 11–20, Aug. 2000.

[5] J. Chomicki, J. Lobo, and S. Naqvi. A Logic Programming Approach to Conflict Resolution in Policy Management. In

Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR '00), pages 121–132, Apr. 2000.

[6] E. de Lara, D. S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based Adaptation for Mobile Computing. In *USENIX Symposium on Internet Technologies and Systems (USITS 2001)*, Mar. 2001.

[7] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat. Model-Based Resource Provisioning in a Web Service Utility. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Mar. 2003.

[8] C. Efstratiou, A. Friday, N. Davies, and K. Cheverst. Utilising the Event Calculus for Policy Driven Adaptation on Mobile Systems. In *Proceedings of the Third International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, June 2002.

[9] B. Ensink and V. Adve. Coordinating Adaptations in Distributed Systems. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, Mar. 2004.

[10] X. Fu, W. Shia, A. Akkerman, and V. Karamcheti. CANS: Composable, Adaptive Network Services Infrastructure. In *Proceedings of the Third USENIX Symposium on Internet Technologies and Systems (USITS '01)*, Mar. 2001.

[11] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer*, 37(10), Oct. 2004.

[12] S. D. Gribble, M. Welsh, R. von Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. Katz, Z. Mao, S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. *IEEE Computer Networks, Special Issue on Pervasive Computing*, 35(4), Mar. 2001.

[13] A.-C. Huang and P. Steenkiste. Building Self-configuring Services Using Service-specific Knowledge. In *Proceedings of the Thirteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-13)*, pages 45–54, June 2004.

[14] J. Huang, R. Jha, W. Heimerdinger, M. Muhammad, S. Lauzac, B. Kannikeswaran, K. Schwan, W. Zhao, and R. Bettati. RT-ARM: A Real-Time Adaptive Resource Management System for Distributed Mission-Critical Applications. In *Proceedings of the 18th IEEE Workshop on Middleware for Distributed Real-Time Systems and Services (RTSS '97)*, Dec. 1997.

[15] H. V. Jagadish, A. O. Mendelzon, and I. S. Mumick. Managing Conflicts between Rules. In *Proceedings of the 15th ACM SIGACT/SIGMOD Symposium on Principles of Database Systems (PODS '96)*, pages 192–201, June 1996.

[16] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, pages 156–171, Dec. 1995.

[17] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen. A Scalable Solution to the Multi-Resource QoS Problem. Technical Report CMU-CS-99-144, Carnegie Mellon University, May 1999.

[18] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken. Specifying and Measuring Quality of Service in Distributed Object Systems. In *Proceedings of the First IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98)*, Apr. 1998.

[19] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, Oct. 1997.

[20] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)*, pages 177–186, Apr. 1998.

[21] V. Poladian, J. P. Sousa, D. Garlan, and M. Shaw. Dynamic Configuration of Resource-Aware Services. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 604–613, May 2004.

[22] S. R. Ponnekanti and A. Fox. SWORD: A Developer Toolkit for Web Service Composition. In *Proceedings of the Eleventh World Wide Web Conference (WWW 2002), Web Engineering Track*, May 2002.

[23] P. Reiher, R. Guy, M. Yarvis, and A. Rudenko. Automated Planning for Open Architectures. In *Proceedings of the Third IEEE Conference on Open Architectures and Network Programming (OPENARCH 2000) – Short Paper Session*, pages 17–20, Mar. 2000.

[24] D. I. Rosu, K. Schwan, S. Yalamanchili, and R. Jha. On Adaptive Resource Allocation for Complex Real-Time Applications. In *Proceedings of the 18th IEEE Workshop on Middleware for Distributed Real-Time Systems and Services (RTSS '97)*, Dec. 1997.

[25] B. Spitznagel and D. Garlan. A Compositional Approach for Constructing Connectors. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA '01)*, Aug. 2001.

[26] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility Functions in Autonomic Systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC'04)*, pages 70–77, May 2004.