

Partitionable Services: A Framework for Seamlessly Adapting Distributed Applications to Heterogeneous Environments

Anca-Andreea Ivan, Josh Harman,* Michael Allen,* and Vijay Karamcheti
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University, New York, NY 10012
{ivan,vijayk}@cs.nyu.edu

Abstract

Several recently proposed infrastructures permit client applications to interact with distributed network-accessible services by simply “plugging in” into a substrate that provides essential functionality, such as naming, discovery, and multi-protocol binding. However, much work remains before the interaction can be considered truly seamless in the sense of adapting to the characteristics of the heterogeneous environments in which clients and services operate.

This paper describes a novel approach for addressing this shortcoming: the partitionable services framework, which enables services to be flexibly assembled from multiple components, and facilitates transparent migration and replication of these components at locations closer to the client while still appearing as a single monolithic service. The framework consists of three pieces: (1) declarative specification of services in terms of constituent components; (2) run-time support for dynamic component deployment; and (3) planning policies, which steer the deployment to accommodate underlying environment characteristics.

We demonstrate the salient features of the framework and highlight its usability and performance benefits with a case study involving a security-sensitive mail service.

1. Introduction

Recent developments in middleware infrastructures have considerably simplified the construction of distributed applications that comprise client programs interacting with one or more network-accessible services. Grid technologies such as Globus [12] and Legion [22] have made possible applications in varied areas such as distributed supercomputing [1, 21], smart instruments [18], and teleimmersion [5],

while web service connectivity and description standards such as HTTP [25], SOAP [26], and WSDL [27] have proven critical to the growth of web commerce. In each case, the infrastructure enables application components to ‘plug in’ into a common substrate, which provides core functionality — naming, discovery, multi-protocol binding, authentication, and authorization.

Although such infrastructures provide location and protocol transparency, much work remains before applications can be truly isolated from the performance and security characteristics of the heterogeneous environments where clients and services operate. To illustrate, consider an application consisting of a mail service accessed by clients from geographically distributed locations across networks with differing bandwidth and security properties. To provide each client with the illusion of a mail server located in its proximity, the application needs to cache server state at various locations, and additionally cope with issues of user authentication, data integrity, and confidentiality in a fashion that is customized to the specific networks under use. Currently, such adaptation is ultimately the responsibility of the application developer, although researchers are beginning to look at infrastructure-level support in certain contexts [7, 3, 6, 18, 2, 13]. However, few proposed techniques are applicable to general-purpose applications, and have rarely taken security considerations into account.

In this paper, we describe a novel adaptation framework, *partitionable services*, which attempts to address these shortcomings. Our framework enables services to be built up from multiple components that can be flexibly assembled to suit the properties of their environment, and facilitates on-demand transparent migration and replication of these components at locations closer to clients while still retaining the illusion of a monolithic service. In the mail application above, the framework automatically determines when and where to place caches of the service and how best to satisfy client expectations of data integrity and con-

*The authors contributed to this work when they were undergraduate students at New York University.

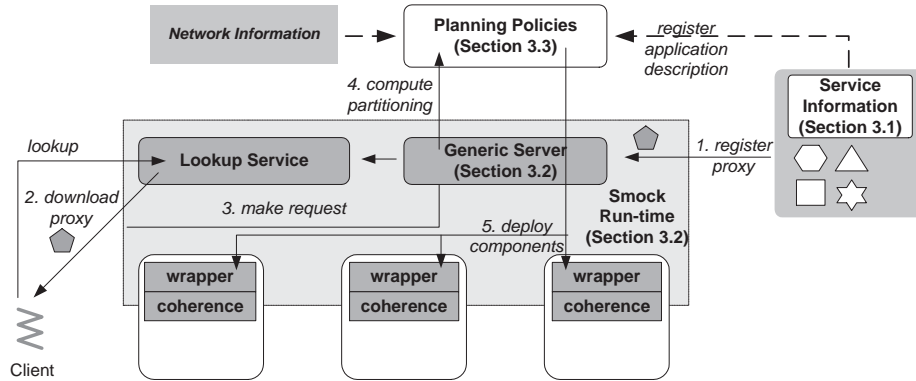


Figure 1. Partitionable services framework.

confidentiality on different networks. To achieve its goals, the framework relies on three pieces: (1) a *declarative specification* of the service components; (2) *run-time system support* for component deployment; and (3) *planning policies*, which steer the deployment by factoring in application structure, network conditions, and client preferences.

The partitionable services framework has been implemented in a Java/RMI and Jini-based infrastructure. To evaluate the costs and benefits of using the framework, which are clearly dependent on the flexibility inherent in the service specification, we describe a case study involving the security-sensitive mail application described above. Our early experience demonstrates that the framework is convenient to use, and is able to successfully adapt to environment characteristics while incurring minimal run time costs.

The rest of this paper is organized as follows. Section 2 overviews the partitionable services framework and introduces the mail service example. Section 3 describes the salient features of the framework and Section 4 details the case study. Related work is discussed in Section 5 and we conclude in Section 7.

2. The Partitionable Services Architecture

Figure 1 illustrates the interaction between different components of the partitionable services framework. The components themselves are described in additional detail in Section 3; here, we briefly describe the timeline of actions enabled by the framework.

The service first registers itself with the framework (Step 1), providing a meta-description of its constituent pieces and procedures for their assembly. As part of this registration, a generic proxy for the service gets uploaded into a lookup service. When a client needs to access the service, it first looks up the service attributes in the lookup service (similar to the Jini infrastructure [24]), downloading the proxy in return (Step 2). This proxy connects to a generic server, forwarding the client request along with supporting

credentials (Step 3). The server passes on the request to a planning module, which considers both the service specification and the current network conditions to come up with a service partitioning that best satisfies the client request (Step 4). To achieve this partitioning, the run-time system of the framework may need to deploy additional components; *wrappers* running on each node facilitate remote installation and *coherence* modules enable application-specific consistency among replicas (Step 5). Once the service components have been installed, the generic proxy at the client replaces itself with a service-specific proxy before returning control to the requesting application.

Mail Service Example Throughout this paper, we highlight different aspects of our framework using the example of a security-sensitive mail service, built using the following components: *mail clients* of differing capabilities, a *mail server* component that can be replicated as desired, and *encryption/decryption* components that ensure confidentiality of interactions between the other components. In addition to traditional mail functionality — user accounts, folders, contact lists, and the ability to send and receive e-mail, our example service allows a user to associate a *sensitivity level* with each message according to its sender or recipient. Each level is associated with an encryption/decryption key pair (one per user) generated at account setup time. The service transparently encrypts messages according to the sender’s sensitivity upon a send, and transforms these messages to those encrypted to the recipient’s sensitivity upon a receive.

When satisfying a client request for this service, one needs to consider several questions: whether the client can benefit from attachment to a (dynamically created) cache of the server, whether the network links provide sufficient security to ensure confidentiality of component interactions, and whether the node being considered for instantiation of a server component can be entrusted with the keys for a specific sensitivity level. As we shall demonstrate, our framework is capable of making such decisions automatically us-

ing only a natural, high-level specification of the service and system conditions.

3. Architecture Components

The automatic deployment of service components in a heterogeneous environment relies upon three pieces: a precise service description, an efficient run-time system, and a planning module.

3.1. Declarative Specification of Services

A central feature of our framework is its ability to adapt to different network conditions by assembling the service from an appropriate combination of components. To support such flexible assembly, the framework leverages a high-level service specification, which merely declares *constraints* on linking one component to another instead of statically specifying linkages.

Figure 2 shows parts of the specification of our mail service,¹ the salient features of which are discussed below.

Interfaces and Properties These define the namespace for the remainder of the specification. *Interfaces* play the same role as in object-oriented languages, serving as the granularity for identifying functionality implemented by the service. *Properties* refer to service-specific parameters that serve as attributes for interfaces and additionally influence component behavior and linkages. The specification identifies allowable values for each property. For example, Figure 2 shows two properties, *Confidentiality* and *TrustLevel*, which can take on a Boolean value or an integer value in the range (1,5) respectively. In general, a property can be defined as a function of other properties.

It is important to note that the framework does not assume any information about the semantics of a given property with respect to the service (e.g., that the *Confidentiality* property refers to whether or not data produced by a component can be deemed confidential); its only concern is with the range of values that can be associated.

Components and Views The core part of the specification identifies the components that make up the service. Like in traditional object-oriented languages, components *implement* interfaces. However, a novel aspect of our application model involves *views*, which can be thought of as customized implementations of a component. Views, originally introduced in [17], can be used to provide only part of the original component’s functionality (*object views*), or contain only part of the original component’s state (*data views*). In both cases, a view needs to be kept consistent

¹Our service specifications use an XML format; however, the examples in this paper are written in a different form to improve readability.

with its original object, a relationship denoted using the `Represents` keyword. A single view definition can additionally be instantiated into multiple component ‘configurations’ at run time; the `Factors` keyword specifies how these configurations can be realized by associating specific values with service properties.

Figure 2 shows four components (`MailClient`, `MailServer`, `Encryptor`, `Decryptor`) and two views (`ViewMailClient`, `ViewMailServer`). `ViewMailClient` exemplifies an object view, which restricts the functionality of the `MailClient`: both support standard send and receive operations, but the latter provides additional features such as access to an address book. `ViewMailServer` is an example of a data view of the `MailServer` component, capable of storing only a subset of the user accounts present in the original. Different configurations of `ViewMailServer` can be realized by setting different values for the `TrustLevel` property. In our example, this influences whether or not messages of a given sensitivity level are sent to or stored in the corresponding `ViewMailServer` component.

Component linkages The primary constraint in the specification governs the linkage between two components. A ‘client’ component C_1 can be connected to a ‘server’ component C_2 only if C_2 implements an interface that C_1 requires. As shown in Figure 2, each component defines a list of interfaces that it implements and those that it requires, also specifying values of properties (either generated or required) for each interface. A description of the latter is deferred to Section 3.3, which refines the simple string-matching based notion of component linkage introduced here to one that also considers compatibility between values of interface properties. Component linkages provide a concise representation for the (potentially large) space of valid service configurations; the latter can be enumerated by constructing all component graphs whose edges correspond to valid linkages.

To consider some examples from Figure 2, the `MailClient` component can connect as a ‘client’ to the `MailServer` component because the latter implements the `ServerInterface` required by the former. Similarly, a `ViewMailServer` component can be connected to a `MailServer` component and an `Encryptor` component can be a ‘client’ for a `Decryptor` component.

Conditions A secondary constraint governs whether a component can be installed at a particular place in the network and is specified using the `Conditions` keyword. Components specify requirements on the environment by requiring service-specific properties to take on certain values. As we discuss in Section 3.3, environment characteristics are translated into properties relevant to the service using a service-supplied external procedure.

<pre> <Property> Name: Confidentiality Type: Boolean Values: T, F </Property> <Interface> Name: ClientInterface Properties: Confidentiality, TrustLevel </Interface> </pre>	<pre> <Property> Name: TrustLevel Type: Interval ValueRange: (1,5) </Property> <Interface> Name: ServerInterface Properties: Confidentiality, TrustLevel </Interface> </pre>	<pre> <Property> Name: User Type: String </Property> <Interface> Name: DecryptorInterface Properties: Confidentiality </Interface> </pre>
--	---	--

<pre> <Component> Name: MailClient <Linkages> <Implements> Name: ClientInterface Properties: Confidentiality = F, TrustLevel = 4 </Implements> <Requires> Name: ServerInterface Properties: Confidentiality = T, TrustLevel = 4 </Requires> </Linkages> <Conditions> Properties: User = Alice </Conditions> </Component> <Component> Name: Encryptor <Linkages> <Implements> Name: ServerInterface Properties: Confidentiality = T </Implements> <Requires> Name: DecryptorInterface </Requires> </Linkages> </Component> <Component> Name: Decryptor <Linkages> <Implements> Name: DecryptorInterface </Implements> <Requires> Name: ServerInterface Properties: Confidentiality = T </Requires> </Linkages> </Component> </pre>	<pre> <View> Name: ViewMailClient Represents: MailClient ... </View> <Component> Name: MailServer <Linkages> <Implements> Name: ServerInterface Properties: Confidentiality = T, TrustLevel = 5 </Implements> </Linkages> <Behaviors> Capacity: 1000 </Behaviors> </Component> <View> Name: ViewMailServer Represents: MailServer <Factors> Properties: TrustLevel = Node.TrustLevel </Factors> <Linkages> <Implements> Name: ServerInterface Properties: Confidentiality = T, TrustLevel = Node.TrustLevel </Implements> <Requires> Name: ServerInterface Properties: Confidentiality = T, TrustLevel = Node.TrustLevel </Requires> </Linkages> <Conditions> Properties: Node.TrustLevel ∈ (1,3) </Conditions> <Behaviors> RRF: 0.2 </Behaviors> </View> </pre>
---	--

Figure 2. Declarative specification (incomplete) of the example mail service.

In Figure 2, both the `MailClient` and the `ViewMailServer` components specify installation conditions. The former requires that the `User` property have the value `Alice` (essentially realizing an access-control list), while the latter requires that the `TrustLevel` property have a value in the range (1,3) for the environment. Informally, this requirement states that a node in the environment must be sufficiently trusted by the service owner before a `ViewMailServer` component can be instantiated there.

Behaviors The final part of the specification conveys information about the resource requirements of a component. Taking the view that a component is a ‘server’ that can satisfy requests from its ‘clients’, several metrics are of interest: (1) the per-request CPU requirement; (2) the number of requests per unit time; (3) the average number of bytes per request and response; and (4) the Request Reduction Factor (RRF), defined as the ratio of requests made along the required linkages in response to a request for one of implemented interfaces. Our specification uses the `Behaviors` keyword to list these component properties, assumed to have been obtained either using profiling or other a priori means. The planning module uses these resource requirements to decide where to instantiate a component so as to optimize a global performance metric.

In Figure 2, the `MailServer` and `ViewMailServer` components show two examples of using the `Behaviors` keyword, specifying that the capacity of the former is 1000 requests/second, and that the RRF of the latter is 0.2. In other words, a `ViewMailServer` component, on average, is able to respond to 80% of its requests based on its own cached state. Although our example provides a static value for RRF, in practice we expect its value to depend on the service properties.

3.2. The Smock Run-time System

the framework relies upon run-time functionality embodied in the Smock infrastructure.² The main pieces of Smock include: (1) a generic proxy and server; (2) wrapper components on each node to facilitate remote component installation; and (3) a coherence module to maintain consistency between replicated components.

Generic proxy and server Service registration simply informs the generic server about the availability of the service and installs a generic proxy into a Jini-like namespace [24]. Clients locate and download the proxy by using an attribute-based lookup service. Requests for service access are sent through the proxy to a generic server, which consults the

²Smock stands for **Secure Mobile Code**. (The “k” was added to create a meaningful word). Smock can be regarded as a layer of software that ‘covers’ the environment.

planning module (Section 3.3) to decide on an appropriate selection and placement of service components. Although requests for a single service are handled in a centralized fashion at the same generic server, note that this is no different from what would have happened anyway in the case of a monolithic service. The framework itself ensures that the generic server does not become a bottleneck by spreading out requests for different services among multiple instances.

Node wrappers Remote component deployment is simplified by the assumption that all nodes have a special environment. Once a component is downloaded on a node, the node wrapper is responsible for initializing it and connecting it to other components, according to the required interfaces specifications. Smock is implemented in Java and benefits from the latter’s support for dynamic class loading, verification, and installation.

Cache coherence layer Smock manages replicated component instances using a directory-based cache coherence protocol. The protocol maintains object consistency at the granularity of views [17]. Coherence actions are triggered based on dynamic *conflict maps*; the latter define when a view conflicts with another and allow expression of a wide range of service-specific weak consistency protocols (including time-driven consistency) necessary for efficient replication in wide-area environments.

The above description reflects the current state of our implementation, which has focused only on run-time aspects that are novel to our framework. In a complete practical system, Smock clearly needs to be integrated with other components that are responsible for fault handling, distributed authentication and authorization (including trust management [10]), and network monitoring [8].

3.3. Planning Policies

The main active component of our framework is the planning module, which is responsible for determining how best to satisfy client requests by instantiating service components at appropriate locations in the network. The planner makes this decision based on two inputs: the service specification described in Section 3.1, and the current state of the network.

From the perspective of the planner, the network is represented as a graph consisting of nodes and links, modeled in terms of their resource characteristics (CPU capacity, bandwidth, latency) and application-independent credentials. The latter refer to properties that are not performance related, e.g., that a particular node does or does not belong to the same administrative domain as the service owner. When planning component deployments for a specific service, the planner first needs to translate these credentials into properties that the service cares about based on

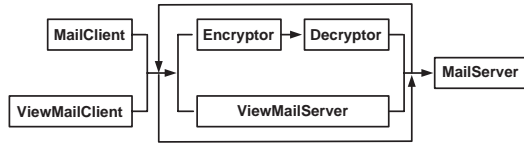


Figure 3. Valid component chains in the mail application.

external service-specific functions. In our mail service example, node and link credentials need to be translated into values of two service properties, *Confidentiality* and *TrustLevel*. Informally, these correspond to whether or not a link/node can maintain confidentiality of component interactions, and the extent to which a node can be trusted.

Our current implementation of the planner makes two assumptions: (i) the network is static, i.e. the set of nodes and links do not change over time, and (ii) the network properties and service component properties remain fixed over the lifetime of a service deployment. In Section 6, we describe future work that will relax these assumptions.

In response to a client request for one or more service interfaces, the planner determines a suitable deployment by logically performing two steps: (1) *finding all valid linkages of service components* that can satisfy the client request, and (2) *mapping these graphs to the network* while ensuring that the mapping satisfies all service constraints. Currently, our planner implementation combines these two steps and exhaustively searches for a deployment that satisfies the constraints described below. For the case where all component graphs are chains, an efficient dynamic programming algorithm is described and evaluated in [13]. More generally, however, applications need to be represented as a directed component graph. To support such applications, we are developing a partial-order based constraint solver modeled after AI planning tools such as IPP [15].

Finding valid linkages The first step, finding valid linkages, is straightforward. The planner starts off with the interface(s) requested by the client, and finds components that implement these interface(s). It then recurses on each of these components by looking at their required interfaces, stopping when it encounters a component without any required interfaces.

Figure 3 shows the valid component chains in our example, given a client request for *ClientInterface*. Any path that originates at either the *MailClient* or *ViewMailClient* component and terminates at the *MailServer* component can satisfy the client request.

Mapping Linkage Graphs The second step, mapping linkage graphs into the network is more involved. Logically, for each linkage graph, the algorithm considers all possible

mappings and discards the ones that do not satisfy one or more of the service constraints. Of the ones that remain, the planner picks the one that optimizes a global objective (maximum capacity, minimum deployment cost, etc.). To validate a mapping, the planner checks three conditions for each pair of linked components:

1. Each of the components can be instantiated in their node environments.
2. The properties of interfaces implemented by the ‘server’ component are *compatible* with those required by the ‘client’ component, given the node/link environment of the linkage.
3. The expected request traffic between the two components does not exceed the capacity of the node/link environment.

Condition 1 above just ensures compliance with the deployment conditions described in Section 3.1. For instance, a *ViewMailServer* component cannot be deployed on a node that has a *TrustLevel* of 0, since this would compromise the security of the internal state of the server.

Condition 2 places additional restrictions on valid linkages by requiring compatibility not only at the interface level but also at the level of properties associated with these interfaces. In our case, this means that the properties of the interface implemented by the ‘server’ component must be a superset of the interface properties required by the ‘client’ component. A complication that must be handled is the environment and its affect on interface properties. To consider an example, the *Confidentiality* property in the mail service example can no longer be ensured if two components are connected across an insecure link.

To cope with this situation, we *explicitly* model the transformation of the implemented interface properties because of the environment via *property modification rules*. Figure 4 shows how the *Confidentiality* property is modified by the environment. Informally, an interface retains a *Confidentiality* property of value ‘T’ only in an environment that has the same value, i.e., is secure. An impact of the rule in Figure 4 is that the planner rejects direct connections between the *MailClient* and *MailServer* components across insecure links, which violate the *Confidentiality*=‘T’ requirement of the *MailClient* component. A clarification: it might appear that our approach is unduly complicated for the simple case of ensuring confidentiality of data transmission across insecure links, which can be accomplished simply through a secure communication channel. The point here is that our approach is generally applicable to properties other than just security, e.g. QoS properties such as delivered video frame rate.

Finally, condition 3 computes the expected load on the involved node(s) and link by scaling the input request rate

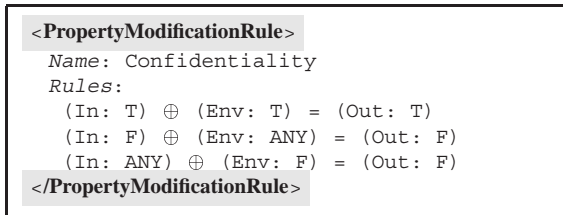


Figure 4. Property modification rules.

with the work performed by the component on behalf of each request (for the node load), and the component’s RRF (for the link load). The planner rejects mappings where the resulting load exceeds the node or link capacity. In our example, this condition results in a preference for the ViewMailServer component in low-bandwidth environments over a direct connection to the MailServer component because of the former’s caching benefits. Note that this decision is based solely on the component requirements and the network properties (computational power of a node, bandwidth of a link), and do not rely upon the semantics of a component’s operation.

4. Case Study: Deploying the Mail Service

To illustrate the deployments produced by the framework and quantify its costs and benefits, we consider a scenario where our example mail service is used by a company to provide e-mail facilities to its members spread across three sites: the main office (New York), a branch office (San Diego), and a partner organization’s site (Seattle).

The network emulating this scenario, which was generated using Boston University’s BRITE tool [19], is shown in Figure 5 and embodies the following constraints: (1) the primary mail server is located in New York; (2) links connecting the three sites are insecure, slow, and of limited bandwidth; and (3) the partner organization nodes (Seattle) are trusted less than those in New York and San Diego. Within each site, the links are considered secure with a fast connectivity of 100 Mbps. The above scenario was realized by building the mail service components described in Section 2. Our current implementation of the framework uses JDK 1.3, Jini 1.2, the XML Winter Pack 01, and the Cryptix JCE implementation. The network topology was emulated using Pentium III (1 GHz) nodes connected using a software router built with the Click modular router infrastructure [16]: traffic shaping components were used to simulate link bandwidth and latency properties.

4.1. Deployments Generated by the Framework

Figure 6 shows the deployments that were produced by the framework given the application specification partially

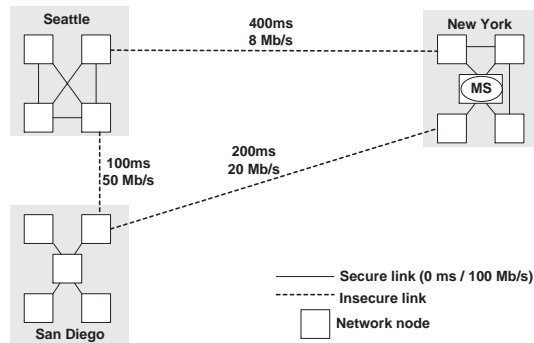


Figure 5. Network topology for the mail service case study.

illustrated in Figure 2.

Client requests in New York result in the deployment of a MailClient component, which connects directly to the MailServer.

Client requests in San Diego result in the deployment of a MailClient, a ViewMailServer, and an Encryptor component in the nodes there, and a Decryptor component in New York. The components are linked together so that the MailClient connects to the ViewMailServer, which links to the MailServer in New York through the Encryptor-Decryptor pair. The ViewMailServer component is instantiated because the planner finds its RRF necessary to traverse the low bandwidth connection between San Diego and New York. The Encryptor and Decryptor components were instantiated because the Confidentiality property would not be respected by a direct connection between the ViewMailServer and MailServer.

Client requests in Seattle result in the deployment of a ViewMailClient and a ViewMailServer component (with a lower trust level), which connects

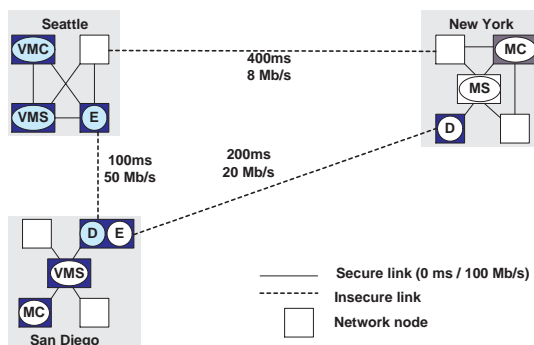


Figure 6. Dynamically deployed components.

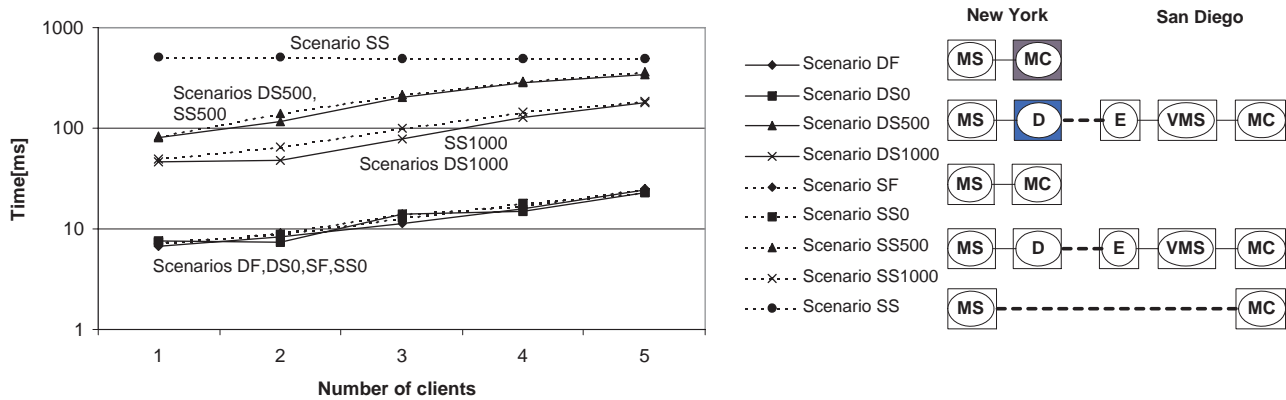


Figure 7. Average client-perceived send latencies incurred by different mail service deployments.

to the ViewMailServer in San Diego through an Encryptor-Decryptor chain as before. The planner chooses to link up the two ViewMailServer components because this results in lower request latency than connecting to the MailServer component in New York.

All of these deployments are realized by interactions between the generic server and node wrappers. The ViewMailServer components are kept coherent with the MailServer component using a protocol that limits the number of unpropagated messages at each replica.

4.2. Performance of the Deployments

To quantify the benefits and costs of using the partitionable services framework, we measured the performance of the nine scenarios shown in Figure 7. Scenario DF corresponds to dynamic deployment over a fast connection for clients in New York, and Scenarios DS0, DS500, DS1000 correspond to the same for clients in San Diego but with different coherence overheads (none, every 500 messages, every 1000 messages) and over a slow connection. Scenarios SF, SS0, SS500, SS1000, SS were hand-generated and model corresponding static scenarios, providing a performance baseline for our dynamic deployments.

Figure 7 shows the average client-perceived send latency for each scenario for 1 up to 5 clients. Each client simulates the behavior of a cluster of users by sending out 100 messages and receiving messages 10 times at the maximum rate permitted by a deployment.

We find that the costs incurred by deploying the mail application as part of the partitionable services framework are negligible compared to the significant gain in user experience. In particular, the plots in Figure 7, which are clustered into four groups — best performing Group 1 (scenarios SF, SS0, DF, and DS0), Group 2 (scenarios SS1000 and DS1000), Group 3 (scenarios SS500 and DS500), and Group 4 (scenario SS) — highlight three key points:

First, the automatically generated dynamic deployments incur negligible additional overhead (virtually indistinguishable in Figure 7) as compared to their static counterparts. Second, the fact that the framework chooses to automatically deploy a caching component before the slow link (scenarios DS0, DS500, and DS1000) results in a substantial performance gain, while retaining the usability of the simplest static scenario, SS, where clients directly connect to the MailServer component unaware of the slow link. And third, comparing the DF scenario with DS0, DS1000, and DS500, we find that the framework approaches the ideal goal of *performance-transparent service access*, enabling remote access costs to be comparable to local access costs to the extent permitted by the underlying cache coherence protocol. The costs for the latter are ultimately determined by the service; however, the framework provides sufficient flexibility to take advantage of relaxed consistency protocols where applicable.

There are a few one time costs not reflected in Figure 7. These include the costs of downloading the proxy, planning, and component deployment and startup. These costs sum up to approximately 10 seconds in the configurations above, but are incurred only at the beginning of the entire process. Our current work is improving the component deployment and setup process to further reduce costs.

5. Related Work

The partitionable services framework builds on our previous work on object views [17], application tunability [4], and CANS [13], which have investigated different aspects of building adaptation-capable distributed applications. Our current work distinguishes itself by pursuing a more general component-based application programming model, similar to that advocated by web-service frameworks such as J2EE [23] and .NET [20], as well as grid-service architec-

tures such as Globus [12] and Legion [22].

Recent work on the Open Grid Services Architecture (OGSA) [12] has explored the integration of web services and grid services, observing that traditional web services can benefit from grid service features dealing with component discovery, resource allocation, authentication and authorization, and lifetime management. The declarative service specifications we propose should be thought of as further refining the WSDL [27] or other XML-based service descriptions to facilitate adaptation (including customization) to heterogeneous environments.

The objectives of our framework are closely related to other recently proposed approaches for QoS-aware deployment of applications in heterogeneous and dynamically changing distributed environments. These approaches can broadly be classified into two categories. The first category, exemplified by the Globus Architecture for Reservation and Allocation (GARA) [11] and Darwin [3], focuses on identifying and reserving appropriate resources in the network to satisfy application requirements, in a sense *adapting the network to the application*.

The second category complements the first by examining how in situations where the network resources should be treated as a given, the application can itself be adapted to achieve desired QoS requirements. The techniques that have been proposed can be further broken down into two classes. The first class comprises systems such as EPIQ [7], ErDos [6], Active Harmony [14], and our own Application Tunability framework [4], which assume that the application structure is more or less fixed and adaptation is achieved by altering the *internal behavior* of one or more of the components (e.g., changing an internal algorithm). The second class of approaches, exemplified by systems such as Active Frames [18], Active Streams [2], Eager Handlers [28], and our CANS [13], has focused on *external behaviors* looking at the adaptation of *data streams* flowing between static application components using *application-specific filters* that can be dynamically introduced and placed at appropriate places in the network.

The partitionable service framework falls into the second class above, but embodies a more flexible application structure than just data streams. Our use of `implements` and `requires` to specify component assembly permits adaptation to affect not only individual component behaviors or the nature of interactions between them, but also which components make up the application and how these are connected. Another notable difference is the focus on adapting to network properties that are not performance related (e.g., security considerations): our framework treats these uniformly by introducing the notion of application-specific properties, specific values for which can be required before a component is instantiated. This last point is related, albeit working at a component granularity, to recent work on

secure program partitioning [9], which uses variable-level security annotations to split programs so that the resulting program complies with various security constraints.

6. Limitations and Future Work

The current implementation of the framework suffers from two limitations, which we intend to address as part of our future work.

First, the framework assumes that properties of the nodes and links of the network where service components are being deployed remain fixed over the lifetime of a deployment. Relaxing this assumption requires that the framework be integrated with network monitoring tools such as Remos [8], which obtain relevant information about the state of the network and communicate it to network-aware application through a well-defined and uniform set of API's. Information about node and link properties (both performance-related and others) obtained by such tools can be communicated to the planner, which determines whether a new deployment (either incremental or complete) is called for. Note that, service redeployment needs to preserve state compatibility between the two configurations and needs to carefully consider the internal state of components as well as any partially processed requests. The framework's cache coherence layer helps with the former issue, and we intend extending techniques that we have developed as part of the CANS [13] system to address the latter issue.

Second, the framework assumes a service-specific way of transforming network attributes into properties of interest to the service. Although such an approach is applicable in several scenarios, ideally one would like to have a service independent mechanism for accomplishing this. One way of achieving this goal is to associate both network and service components with different types of *credentials*, whose namespace refers to the properties of interest in each case. Transforming properties in one namespace into properties in another then becomes a simple matter of issuing a different kind of credential, which *delegates* to one all of the privileges associated with the other. This perspective motivates the use of trust-management infrastructures traditionally employed to establish security policies in systems spanning multiple administrative domains. As part of other work in our research group, we have developed a decentralized trust management mechanism, dRBAC [10], which we intend to adapt for use in our framework. Using dRBAC, network properties, service properties, translation between the two, and deployment requirements can all be expressed using a common language. Additionally, the dRBAC implementation takes responsibility for continuous monitoring of credential validity and thereby provides a convenient way of coping with changing network and/or service properties.

7. Conclusions

In this paper, we have demonstrated that the partitionable services framework can enable efficient and transparent access to services by flexibly assembling and dynamically deploying components into the network, starting from high-level descriptions of service and network behavior, and client QoS requirements.

The component-based view of services, complemented with automatic tools for their deployment and management, offers a promising approach for improving usability of network-based services in diverse network environments and under varied usage scenarios.

Acknowledgements

This research was sponsored by DARPA agreements F30602-99-1-0157, N66001-00-1-8920, and N66001-01-1-8929; by NSF grants CAREER:CCR-9876128 and CCR-9988176; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Labs, SPAWAR SYSCEN, or the U.S. Government.

References

- [1] G. Allen and E. Seidel. Cactus Computational Toolkit. www.cactuscode.org, 1998.
- [2] F. Bustamante and K. Schwan. Active Streams: An Approach to Adaptive Distributed Systems. *Proc. 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, 2001.
- [3] P. Chandra. Darwin: Customizable Resource Management for Value-Added Network Services. *Sixth IEEE Intl. Conf. on Network Protocols (ICNP)*, 1998.
- [4] F. Chang and V. Karamcheti. A Framework for Automatic Adaptation of Tunable Distributed Applications. *Cluster Computing*, 4:49–62, 2001.
- [5] J. Leigh et al. Adaptive Networking for Tele-Immersion. *Proc. of the Immersive Projection Technology/Eurographics Virtual Environments Workshop (IPT/EGVE)*, 2001.
- [6] J. Sydir et al. QoS Middleware for the Next-Generation Internet. *NASA/NREN Quality of Service Workshop*, 1998.
- [7] M. Shankar et al. An End-To-End QoS Management Architecture. *Proc. of the 5th IEEE Real-Time Technology and Applications Symposium*, 1999.
- [8] P. Dinda et al. The Architecture of the Remos System. *Proc. of 10th IEEE Symposium on High-Performance Distributed Computing (HPDC)*, 2001.
- [9] S. Zdancewic et al. Untrusted Hosts and Confidentiality: Secure Program Partitioning. *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.
- [10] E. Freudenthal et al. dRBAC: Distributed Role-based Access Control for Dynamic Coalition Environments. *To appear in 22nd Intl. Conf. on Distributed Computing Systems (ICDCS)*, 2001.
- [11] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. *Intl. Workshop on Quality of Service*, 1999.
- [12] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, <http://www.globus.org/research/papers.html>, 2002.
- [13] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. CANS: Composable, Adaptive Network Services Infrastructure. *3rd USENIX Symp. on Internet Technologies and Systems*, 2001.
- [14] J. K. Hollingsworth and P. J. Keleher. Prediction and adaptation in active harmony. *Cluster Computing*, 2(3):195–205, 1999.
- [15] J. Koehler. Planning under resource constraints. In *European Conf. on Artificial Intelligence*, pages 489–493, 1998.
- [16] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.
- [17] I. Lipkind, I. Pechtchanski, and V. Karamcheti. Object Views: Language Support for Intelligent Object Caching in Parallel and Distributed Computations. *Proc. of ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 447 – 460, 1999.
- [18] J. Lopez and D. O’Hallaron. Support for Interactive Heavy-weight Services. *Proc. of the 10th Symposium on High Performance Distributed Computing (HPDC-10)*, 2001.
- [19] A. Medina and I. Matta. BRITE: A Flexible Generator of Internet Topologies. Technical Report 2000-005, Boston University, 21, 2000.
- [20] Microsoft Corporation. Microsoft .NET. <http://www.microsoft.com/net/default.asp>.
- [21] NASA. Overflow-D2. <http://halfdome.arc.nasa.gov/cfd/cfd4/>.
- [22] A. Natrajan, M. A. Humphrey, and A. S. Grimshaw. Capacity and Capability Computing using Legion. *Proceedings of the 2001 International Conference on Computational Science (ICCS)*, 2001.
- [23] Sun Microsystems. Java 2 Enterprise Edition. <http://java.sun.com/j2ee>.
- [24] Sun Microsystems. Jini Architecture Specification Version 1.1. <http://www.sun.com/jini/specs/jini1.1.html>.
- [25] World Wide Web Consortium. RFC 2616: Hypertext Transfer Protocol – http/1.1. <http://www.w3.org/protocols/>, 1999.
- [26] World Wide Web Consortium. Simple Object Access Protocol (SOAP) 1.1. <http://www.w3.org/tr/soap/>, 2000.
- [27] World Wide Web Consortium. Web Services Description Language (WSDL) 1.1. <http://www.w3.org/tr/wsdl>, 2001.
- [28] D. Zhou and K. Schwan. Eager Handlers - Communication Optimization in Java-based Distributed Applications with Reconfigurable Fine-grained Code Migration. *3rd Intl. Workshop on Java for Parallel and Distributed Computing (held in conjunction with the IPDPS 2001)*, 2001.