WS-Naming: Location Migration, Replication, and Failure Transparency Support for Web Services

Andrew Grimshaw

Mark Morgan

Karolina Sarnowska

University of Virginia, Department of Computer Science

Abstract:

Naming transparencies, i.e., abstracting the name and binding of the entity being used from the endpoints that are actually doing the work, are used in distributed systems to simplify application development by hiding the complexity of the environment. In this paper we demonstrate how to apply traditional distributed systems naming and binding techniques in the Web Services realm. Specifically, we show how the WS-Naming profile on WS-Addressing Endpoint References can be used for identity, transparent failover, replication, and migration. We begin with a discussion of the traditional distributed systems transparencies. We then present four detailed use cases. Next, we provide brief background on both WS-Addressing and WS-Naming. Finally, we show how WS-Naming can be used to provide transparent implementations of our use cases.

1 Introduction

Most applications bring with them a number of "non-functional" characteristics that are critical for success. For example, the time required to perform an operation, the strength of the authentication mechanism used, availability, the integrity of the data, or the application's ability to continue to function in the presence of faults. These characteristics are often referred to as Quality of Service (QoS), Quality of X (QoX), or Service Level Agreement (SLA) properties. Users, whether they are end-users or an organization that uses the application, want instantaneous response, fool-proof security, absolute data integrity, and non-stop performance. Achieving the desired service level is often difficult or impossible, and trade-offs usually need to be made.

One of the jobs of the systems architect is to provide mechanism with which to develop application implementations that meet the non-functional requirements. Implementing the service levels in single host environments is known to be challenging and responsible for a rich literature [8, 10, 12], e.g., MAPE loops [24]. A distributed, service oriented architecture such as a Web Services based Grid presents significant additional challenges.

The challenges arise out of the distributed nature of Grids: there may be thousands to millions of Web Service containers, in many different mutually distrustful organizations, scattered across the globe, running on hardware that may fail, running applications and services written by thousands of different organizations. To paraphrase Butler Lampson,

"a distributed system is a system in which a machine I've never heard of fails, and I can't get any work done."

If Web Services based Grids are to be successful, standard mechanism must be developed that provides the application developer the ability to cope with the complexity of the underlying environment. Furthermore, these mechanisms should not be monolithic and opaque. Instead, the mechanisms should be "stacked" in the sense that a range of mechanism is available, and that the programmer can choose appropriate mechanism and not "pay" for more than they want. These mechanisms should range from high-level capabilities that take requirements specifications and attempt to enforce those requirements (relieving the programmer from the burden of dealing with them), to simple low-level mechanisms that can be used and combined in a variety of ways by the application programmer [20]. Thus, the application developer has the flexibility to either rely on high-level mechanism, or dive down the stack for a custom solution.

One of the often used mechanisms in distributed systems over the years to simplify application development and deal with the complexity of the environment is the use of multi-level naming schemes to provide naming transparency [5, 15, 17, 18, 22, 23]. Examples of multi-level naming schemes are numerous. For example, Unix path names map to inodes. The top level name is a human readable string that often has semantic meaning to the user, e.g., /bio/data/sequence1. The inode number, e.g., 1297, rarely does. Similarly DNS names are mapped to IP addresses: mail.cs.virginia.edu is meaningful, while 128.143.137.19 is not. In both of these two-level naming scheme examples a human readable name maps to an address.

In three level naming schemes human names are first mapped to abstract names. Abstract names are usually location independent and intentionally opaque to the client. Before a client can communicate with a resource named via an abstract name the abstract name must first be resolved, or bound to an address. The advantage of a three level naming scheme is that the client can work with human names and have the infrastructure manage the binding logic.

In this paper we demonstrate how traditional distributed system naming and binding techniques can be applied to Web Services. Specifically we show how the WS-Naming [7] profile on WS-Addressing [4, 9] Endpoint References (EPRs) can be used for transparent failover, replication, and migration. WS-Naming, and naming transparencies in general, provide a mechanism for realizing a variety of use cases.

The remainder of this paper is organized as follows. We begin with a background discussion on distributed systems, the classic transparencies, and the definition of a number of terms. In section 3 we present four use cases that motivate WS-Naming. These use cases, along with the collective experiences of the WS-Naming working group of the Open Grid Forum (OGF), drove the requirements developed in section 4. Section 5 presents the WS-Naming profile. Section 6 illustrates how WS-Naming can be used to realize the use cases. In section 7 we examine WS-Naming with respect to the requirements laid out in section 4. In section 8 we describe our implementation of WS-Naming. We conclude with a summary and planned future work.

2 Background and Related Work

There is an extensive body of related work from distributed systems and WS-Naming does not claim to be innovative with respect to implementing a two level naming and binding scheme that supports location, migration, failure, and other transparencies; nor are the use cases particularly novel. Migration transparency has been around for some time in systems such as Chorus [19], Amoeba [22], AFS/Coda [21], Sprite [15], the V System [5], Plan 9 [17], Emerald [11], and many others. Similarly, replication for both performance [6] and availability has been used over the years.

WS-Naming takes the lessons learned in the 80's and 90's about distributed systems and applies them in a standards-based context – Web Services. By extending the basic Web Services addressing model with identities and a re-binding mechanism, WS-Naming provides Web Services developers with a powerful set of tools. The mechanism is based on over twenty years of distributed systems work and experience by the community.

Distributed systems research has a rich literature [11-34] in which virtualization of resources and objects figures prominently. This virtualization results in a "transparency". Nine of these show up repeatedly and have been called the "nine golden transparencies". The transparencies are: access, location, failure, heterogeneity, migration, replication, concurrency, scaling, and behavioral. Typically, these transparencies come into play in while accessing remote services or objects. The goal is to hide details from the programmer unless those details are of particular interest for a given application domain.

Four of these, location, failure, migration, and replication transparency are of particular interest here. They are defined as follows.

Location. The caller need not know where the resource is located -- California or Virginia, it makes no difference.

Failure. If the resource fails, the caller is unaware. Even with failure, the requested service or function is performed, the resource restarted, etc.

Migration. The caller need not know whether a resource has moved since they last communicated with it.

Replication. Is there one resource or many resources behind the name? The caller need not know.

Naming as a means for providing transparency has been used extensively both in the earlier cited projects, and in standards efforts over the years. The best known is the Domain Name Service (DNS) that maps strings to IP addresses. DNS is clearly a successful standard. Its has some significant limitations. Most importantly it was not designed to support a highly dynamic binding environment where the mappings could change rapidly. More recent naming schemes include Life Science IDentifiers (LSIDs) [3]from OMG and Handle.net from CNRI [2]. LSIDs share many of the same goals and

objectives as WS-Naming[2], but were designed do not fit well in the context of the existing Web Services infrastructures. LSIDs must always be resolved, requiring the clients to be LSID aware. The same applies to Handle.net handles, as well as a licensing model that made their adoption difficult for many commercial enterprises.

Before we move on we'd like to define a few terms for the purposes of the paper.

Resource – A resource is a namable entity that accepts a set of method calls specified in an IDL such as WSDL. A resource may be an instance of some class or template (itself a resource) – but this is not required. How resources come into existence is not an issue for the purposes of naming. Resources may have metadata or attributes associated with them. The distinction between stateful and stateless services is irrelevant in for us.

Abstract resource name: an abstract name should not rely on any location, type, implementation, or cardinality information being embedded in the name. This ensures that the abstract name can persist across resource migration, container restart, etc.

Sameness. Two abstract names refer to the "same" resource if it is not possible to distinguish between the two resources given arbitrary and equivalent message histories. Alternatively if the semantics of the service define "sameness" differently the service semantics set the standard for that service. If two abstract names are "*aliases*" if the abstract names are different yet they refer to the "same" resource. In other words, from the outside it is not possible to tell them apart. Similarly, two different resource addresses may refer to the "same" resource.

Human names. The two most frequently used examples of human names are paths such as /home/smith/datafile, and properties (a.k.a. attributes and metadata), e.g., "invoice where modification date=9-15-03 and value<$45.00".

Binding: The mechanism that "binds" abstract names to resource addresses, i.e., given an abstract name the binding scheme returns a resource address that can be used to communicate with the resource.

Bind time: The point at which an abstract name is bound to a resource address. Note that this can happen at many different times, e.g., at compile time, at program load time, at first use in a program, on failure of an old binding, and/or on every use. For all but the last case, on every use, we assume that the binding may be cached somehow in the callers context. In some schemes, binding can be an expensive operation, thus there is a trade-off in bind time decisions between performance and dynamics.

3 Driving Use Cases

While existing Web Services best practices support heterogeneity, concurrency and behavioral transparency, the use of abstract naming mechanisms can provide a

[2] LSID had its beginnings in the Global Grid forum as the SGNP – Secure Grid Naming Protocol, but was moved to the I3C and renamed and re-engineered as LSID. It moved to the OMG when the I3C was dissolved.

[4] How the state of the failed instance is kept synchronized with the replica is not the issue here. There are many well-known techniques including, periodic checkpoints, message logging, etc. The important fact is that naming facilitates this process.

framework for realizing several additional important transparencies – in particular migration, location, replication, and failure transparency. In this section we describe five real world use cases that highlight the importance of supporting these additional transparencies.

Caching of results. It is often said that the three most important words in distributed systems are cache, cache, and cache. This is particularly true for wide-area systems where network latencies can easily exceed 50 milli-seconds. While it is not appropriate to cache the results of some services, for others, such as directory or file services, caching can provide significant performance benefits. In order to maintain a cache it is necessary to have some sort of cache key that can be used to uniquely identify each cache entry. A challenge in Web Services is that WS-Addressing EPRs by themselves cannot be compared, and, thus cannot be used as a cache key. Another means is required to determine if a new invocation on a resource using some EPR is directed at the same resource as a previous call using another EPR.

Migrate closer to active users. Suppose that a client application is making intense use of a resource that is physically located far away – for example an application in California that reads and modifies a shared file resource currently residing in New York. In this configuration, the application may be unnecessarily suffering from poor performance due to high network latency. One would like to be able to migrate the file resource from New York to California without any service interruption to other users that need access to the shared file. Thus, migration provides a mechanism for meeting performance SLTs.

Migrate away from failing or overloaded systems. Consider a service or resource executing on a host that is heavily loaded in some way – for example, the host CPU is overloaded or the network into the host is flooded. One would like to migrate the service to another host without interrupting the service and without disrupting on-going interactions with this and other services and resources. Similarly, it may be known that a host is going to "go down" soon, perhaps because of maintenance, or maybe problems with the physical environment (power shortage, air conditioning failure, etc.). Once again, we want to migrate the service to another location without interrupting on-going interactions. Thus, migration can be used to meet performance, availability and reliability SLTs.

Recovery from a failed resource. Consider a stateful resource that has failed (due to a hardware failure, a software failure, etc.) and needs to be restarted – possibly on a different physical resource. One wants to be able to "migrate" the resource instance to a different location or machine while minimizing interruptions for accessing the resource .[4] Thus, migration can support failure masking and be used to meet reliability SLTs.

Replica management and usage. A resource may have multiple "back-end" endpoints that can each perform its services and one would like to dynamically select which replica to use. For example, one replica may be closer to the end-user than another (in network terms) or one replica may offer better QoS in some dimension (e.g., performance). Replication can be used to meet performance, reliability, and availability SLTs.

4 WS-Naming Requirements

After almost a year of discussion of the use cases and requirements in both face-to-face meetings and teleconferences, the OGSA-Naming Working Group in the OGF (then called the Global Grid Forum) developed the following set of required and desired properties for a Web Services naming scheme. There was not complete agreement on some of the requirements. In particular the desirability of aliases was hotly debated. Below we list of requirements with a brief description of each.

Free. The Grid community expects open software license with no license fee.

Compatible – The scheme must work with existing WS tooling that uses EPRs.

Unique – If two names are the "same", they refer to the "same" resource. Further, names must be unique in space and time, and, therefore, names cannot be reused.

Comparable – Given two different names, it is possible to determine equality, without having to contact either the resource or some other third party.

Persistent – The name for a resource is valid as long as the resource exists.

Location portable – An abstract name can be used anywhere and will refer to the same resource from any context. This does not necessarily mean messages will be routable to the resource from any location.

Support the usual transparencies – Location, migration, failure, and replication.

Extensible – To accommodate future requirements the naming scheme must be extensible.

High-performance – Low performance naming schemes prevent scalability and usability. If the scheme is too slow, it will not be used, thus defeating the purpose.

Dynamic binding – Binding an abstract name to an address is not static, i.e., it can change. The speed at which re-binding takes place is critical – if it takes hours to rebind, then resource mobility is highly constrained.

Scalable binding – We expect the scale of future distributed systems to be very large. The binding scheme must scale with the system. Often this can be achieved with caching. However, caching must be balanced against the need to have dynamic bindings.

Language neutral. Some naming schemes were developed specifically to support particular programming languages.

Concurrent name generation. This is an aspect of scalability. It must be possible to partition the name space and generate names concurrently.

Large name-space. Running out of names is not acceptable.

In addition to the required characteristics, there are several desirable characteristics.

Authentication – Mutual authentication between two named resources without requiring a trusted third party has advantages, particularly with respect to scalability.

Widely adopted – The usefulness of a naming scheme increases when it is widely used.

Not require trust – Schemes that do not require a trusted third party are, all else being equal, preferred over those that do require a trusted third party.

Human readable and printable. Often it will be necessary to print a name.

Finally, there was a debate over whether aliases would be permitted. In other words, can a resource have two different names? No consensus was reached to prohibit aliases, therefore, aliases were allowed.

We will examine how well WS-Naming meets the goals in section 6.

5 WS-Naming – A Profile on WS-Addressing

WS-Addressing [4] is a specification from the W3C which describes, among other things, a schema type called an *EndpointReferenceType* (EPR). Clients and services use these EPRs to identify target Web Wervice resources by embedding addressing information contained in the EPR into SOAP message headers. This EPR schema includes fields for identifying the target address (URI) of the desired service, opaque referencing information which services may use to further identify session data, and metadata information which can be used by clients as hints describing various aspects of the target Web Service or Web Service resource. EPRs are the most widely used mechanism for referring to Web Service endpoints.

```
<wsa:EndpointReference
        xmlns:wsa=”http://www.w3.org/2005/08/addressing”
        <wsa:Address>http://tempuri.org/example_application</wsa:Address>
</wsa:EndpointReference>
```

Figure 1. A very simple EPR with an Address element.

WS-Naming was developed to address two short-comings of WS-Addressing. First, EPRs cannot be compared against each other in any canonical way to determine if they refer to the “same” endpoint. Indeed, the specification explicitly states that EPRs cannot be compared.

Second, given the way many WS-Addressing implementations work, an endpoint cannot migrate. To understand why, we need to look at the *wsa:Address* field of the EPR. The *wsa:Address* field is a URI. Technically, this URI could have a location transparent abstract string – in practice however it does not. Every WS-Addressing implementation we have encountered uses a URL with either an IP address or a DNS hostname. Thus, the endpoint is pinned to a particular location at the time it is minted[7].

The authors of WS-Naming wanted a profile on WS-Addressing that would address these two concerns, the requirements presented earlier, and be 100% compatible with existing practice regarding the use of WS-Addressing endpoints. This is critical; EPRs that are

[7] There are tricks you can play with DNS to move endpoints around, particularly in a machine room. However, these do not work well across domains and break the advantages of the cache.

WS-Names MUST be consumable by clients that are completely unaware of WS-Naming, and that assume the *wsa:Address* field is a URL.

WS-Naming describes two extensibility profiles on the standard WS-Addressing specification whereby target service endpoints add additional information to their WS-Addressing EndpointReferenceType's metadata element; namely an endpoint identifier element (EPI) that serves as a globally unique (both in space and time) abstract name for that resource, and a list of zero or more *resolver* EPRs.

5.1 End Point Identifiers (EPIs)

The endpoint identifier element gives clients a way of identifying and comparing addressing endpoints without requiring them to communicate with those endpoints. EPIs are IRIs [RFC 3987] that are contained as elements in the metadata section of an EPR. From the specification, EPIs must satisfy three requirements:

- An EndpointIdentifier MUST uniquely identify the same endpoint in both space and time.
- An EndpointIdentifier MUST conform to IRI syntax [RFC 3987].
- For two equal EndpointIdentifiers (as defined by RFC 3987), a client MAY assume that the two EndpointIdentifiers refer to the same endpoint.

The Endpoint Identifiers are "abstract" in that the client should not infer any property (e.g., type, location) from inspection of the EPI. Clients should treat EPIs as opaque.

The global uniqueness in both space and time requirement can be achieved by a number of means. Implementers are free to choose any name generation scheme that they wish to use provided the scheme generates unique names. Some options include various combinations and hashes of public keys, MAC addresses, generated IP address (not current Web Service endpoint address which may change), timestamp, random number, etc. In particular, the authors of this document recommend that name generators refer to RFC 4122 [RFC4122], which gives a motivation for and description of UUIDs or Universally Unique IDs. One may also choose to acquire a name from an existing naming authority.

Symmetry in endpoint identifier equality is not required. If two EPIs are not bit-wise equal, no conclusions can be drawn as to whether or not they refer to different endpoints.

Figure 2 gives an example of an EndpointIdentifier in an Endpoint Reference.

```
<wsa:EndpointReference
        xmlns:wsa="http://www.w3.org/2005/08/addressing"
        xmlns:naming="http://schemas.ogf.org/naming/2006/08/naming">
        <wsa:Address>http://tempuri.org/example</wsa:Address>
        <wsa:Metadata>
                <naming:EndpointIdentifier>
                        urn:guid:B94C4186-0923-4dbb-AD9C-39DFB8B54388
                </name:EndpointIdentifier>
        </wsa:Metadata>
```

Figure 2. EndpointIdentifier in EPR

5.2 Resolvers

EPRs that follow the WS-Naming extensions may contain one or more "resolver elements" within the wsa:Metadata element. Each "resolver element" identifies a resolver service as well as information that that resolver service needs to uniquely identify the target resource. Even though we typically refer to resolution in WS-Naming generically as a single operation, resolvers actually come in two flavors: the *EndpointIdentifierResolver* porttype resolves from an EPI while the *ReferenceResolver* porttype resolves from an existing EPR. Therefore, depending on the resolver porttype the "resolver element" is really either a naming:ReferenceResolver element or a naming:EndpointIdentifierResolver. The pseudo-interfaces (from the specification) are shown in below.

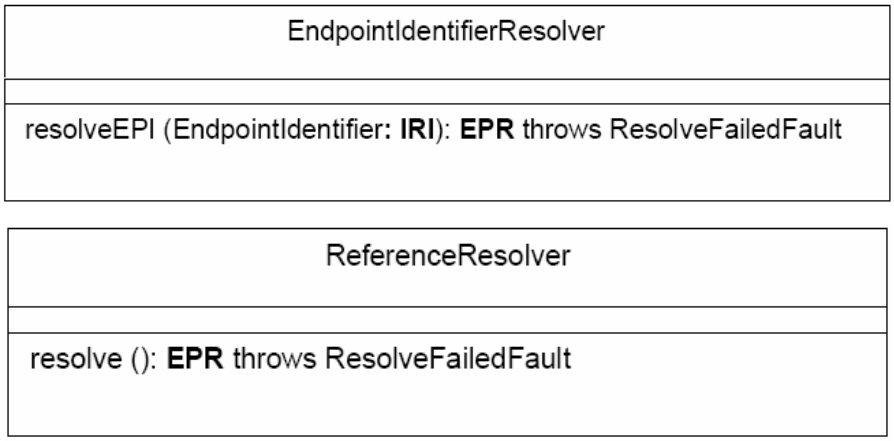

Figure 3. Pseudo-interfaces for the resolution porttypes.

Clients may use the embedded resolver information within a target EPR to call a resolver and obtain a new binding (another EPR) for the target resource. For example, clients attempting to communicate with stale or invalid endpoint references can use a resolver to obtain new, up-to-date, bindings. Note, however, that clients are free to choose how they obtain endpoint references – they are *not* required to use the resolvers provided in the EPR. Any means for resolving EPIs or stale EPRs may be tried at the discretion of the client.

```
<wsa:EndpointReference
        xmlns:wsa="http://www.w3.org/2005/08/addressing"
        xmlns:naming="http://schemas.ogf.org/naming/2006/08/naming">
        <wsa:Address>http://tempuri.org/example_application</wsa:Address>
        <wsa:Metadata>
                <naming:EndpointIdentifier>
                        urn:guid:B94C4186-0923-4dbb-AD9C-39DFB8B54388
                </naming:EndpointIdentifier>
                <naming:ReferenceResolver>
                        <wsa:Address>http://tempuri.org/resolver1</wsa:Address>
                </naming:ReferenceResolver>
                <naming:EndpointIdentifierResolver>
                        <wsa:Address>http://tempuri.org/resolver1</wsa:Address>
                </naming:EndpointIdentifierResolver>
                <naming:ReferenceResolver>
                        <wsa:Address>http://tempuri.org/resolver2</wsa:Address>
                </naming:ReferenceResolver>
        </wsa:Metadata>
</wsa:EndpointReference>
```

Figure 4. The EPR contains both an EndpointIdentifier that uniquely identifies the endpoint, and multiple ReferenceResolvers that can be used to rebind to the identified endpoint if communication fails.

```
<wsa:EndpointReference
        xmlns:wsa="http://www.w3.org/2005/08/addressing"
        xmlns:naming="http://schemas.ogf.org/naming/2006/08/naming">
        <wsa:Address>http://tempuri.org/example_applocation</wsa:Address>
        <wsa:Metadata>
                <naming:ReferenceResolver>
                        <wsa:Address> http://tempuri.org/resolver1 </wsa:Address>
                        <wsa:ReferenceParameters>
                                <naming:EndpointIdentifier>
                                        urn:guid:8733111B-84FA-4da8-89FE-
                                417932B3B92C
                                </naming:EndpointIdentifier>
                        </wsa:ReferenceParameters>
                        <wsa:Metadata>
                                <naming:EndpointIdentifier>
                                        urn:guid:55AD06F6-2F35-409a-9DCE-
                                E5F304E557AA
                                </naming:EndpointIdentifier>
                                <naming:ReferenceResolver>
                                        <wsa:Address>
                                                http://tempuri.org/resolver_resolver1
                                        </wsa:Address>
                                </naming:ReferenceResolver>
                        </wsa:Metadata>
                </naming:ReferenceResolver>
        </wsa:Metadata>
        <naming:EndpointIdentifier>
                urn:guid:B94C4186-0923-4dbb-AD9C-39DFB8B54388
        </naming:EndpointIdentifier>
</wsa:EndpointReference>
```

Figure 5. Reference Resolvers are WS-Addressing EPRs themselves and as such can be fully compliant WS-Naming endpoints including both EndpointIdentifiers and their own Reference Resolvers.

6 Goals Redux

The WS-Naming specification team set out to meet eighteen goals presented in section 4. Below we indicate how well WS-Naming meets those goals.

Goal	WS-Naming	Goal	WS-Naming
Free	Yes.	Scalable	Yes.
Unique	Yes.	Language neutral	Yes.
High Performance	Same performance	Concurrent name	Yes.

	as Web Services..	generation	
Comparable	Yes.	Authentication.	No. Yes, when used with WS-Secure Addressing.
Persistent	Yes.	Widely adopted.	Not yet.
Location portable	Yes.	No required trust	Yes.
Extensible	Yes	Human readable	Barely.
Compatible	Yes.	Aliases	Yes.
Dynamic binding	Yes.		

7 Examples

We next demonstrate how WS-Naming, in conjunction with a Resolver service, is used to support migration and failure transparency. In both cases we will assume that the service in question is an OGSA-ByteIO [1] service "instance". OGSA-ByteIO includes porttypes that support *libc*-like file operations, e.g., *read*, *write*, *append*, etc. While we assume ByteIO here – the techniques apply to a wide variety of services.

We will assume a file **F** with a unique identity **F-EPI**. Thus, when we say "write to **F**" what we mean is to send a SOAP message to the EPR that we have for **F**. Also, to simplify the discussion (and avoid endlessly tedious XML), we will use a pseudo-schema for the WS-Naming EPRs to eliminate unnecessary detail.

7.1 Migration

The first example is migration. Suppose that **F** is initially contained in a hosting environment **H1** on physical host *hadrian.cs.virginia.edu* at port *1660*, and a client **C** is using **F**. Further assume a resolution service with an EPR **S**. It is assumed that **S** always knows where **F** is located. How **S** knows this is out of scope, though one can easily imagine (and implement) migration services that update **S** whenever **F** is migrated. Finally suppose that **F** is moved from **H1** to **H2** on physical host *cache.sdsc.edu* in California at port *2222*.
At the beginning, the EPR for **F** is

<wsa:EndpointReference

<wsa:Address> http://hadrian.cs.*virginia*.edu:1660 </wsa:Address>

<EndpointIdentifier> F-EPI </EndpointIdentifier>

<ReferenceResolver> S </ ReferenceResolver>

</wsa:EndpointReference>

Before **F** migrates, **C** reads and writes **F** using the *hadrian* address. At some point afterwards, the migration occurs.

How the migration is accomplished is again, out of scope. However, in the case of ByteIO files, it is very straightforward. A mechanism such as *scp* or *gridFTP* can be used

to migrate the physical bits of **F**, and **H2** can be set to know where **F** is located in the local file system.

Once the migration starts, **H1** no longer accepts messages for **F**. Instead it returns a *bad address* fault to **C** when **C** reads or writes **F**. **C** will also receive a *bad address* fault if *hadrian* has died or becomes partitioned from the network.

In either case, on receipt of the *bad address* fault, **C** examines the EPR of **F** to determine if it contains a ReferenceResolver EPR (RR-EPR). If it does not, then no further actions can be taken, and the Web Services client stub returns a fault to the application. If there is an RR-EPR **S**, **C** calls the Resolve method on **S** to acquire a new binding for **F**. **S** returns **C** the new EPR for **F** shown below.

<wsa:EndpointReference

<wsa:Address> http://cache.sdsc.edu:2222 </wsa:Address>

<EndpointIdentifier> F-EPI </EndpointIdentifier>

<ReferenceResolver> S </ ReferenceResolver>

</wsa:EndpointReference>

C then calls **F** using the new EPR and the application resumes. The application layer is unaware of the migration that took place.

An important idea to keep in mind is that how the migration is accomplished, why **F** is migrated, when **F** is migrated, where **S** is located, or how **S** knows about the migration is completely out of scope in the specification, and is transparent to the client. This permits a wide variety of different implementations without exposing these details to the client.

7.2 Highly-Available ByteIO Service

Our second example is a highly available ByteIO service implemented via a primary copy mechanism. The details of the replica interactions are out of scope, as are the semantics of write operations with respect to failure (e.g., if the client receives a reply from a write is it guaranteed that the write will survive all possible subsequent failures?). These are obviously important questions, and there are a number of different choices and implementation options available. What we are interested in is the client experience – not the semantics.

Once again assume a ByteIO **F**, a client **C**, two hosting environments **H1** and **H2**, and a resolution service **S**. In this case, however, there are two ByteIO endpoints, $\mathbf{F_p}$ and $\mathbf{F_s}$, the primary and secondary copies of **F** respectively. Further assume that $\mathbf{F_p}$ is on **H1** and that $\mathbf{F_s}$ is on **H2**. Thus the EPR of $\mathbf{F_p}$ is

<wsa:EndpointReference

<wsa:Address> http://hadrian.cs.virginia.edu:1660 </wsa:Address>

<EndpointIdentifier> F-EPI </EndpointIdentifier>

<ReferenceResolver> S </ ReferenceResolver>

</wsa:EndpointReference>

And the EPR of $\mathbf{F_s}$ is

<wsa:EndpointReference

<wsa:Address> http://cache.sdsc.edu:2222 </wsa:Address>

<EndpointIdentifier> F-EPI </EndpointIdentifier>

<ReferenceResolver> S </ ReferenceResolver>

</wsa:EndpointReference>

Initially **C** has the primary copy EPR of **F**, $\mathbf{F_p}$. File operations proceed normally against $\mathbf{F_p}$ until $\mathbf{F_p}$ fails, either because the host **H1** it is on fails, **H1** refuses access (perhaps due to overload) or **H1** and **C** become disconnected. Reguardless of the cause, **C** will receive a *bad address* fault. At this point the flow is exactly the same as in the migration case. The client stub code in **C** checks for a ReferenceResolver EPR in $\mathbf{F_p}$. If there is an RR-EPR **S**, **C** contacts **S** for a new EPR. **S** returns $\mathbf{F_s}$, and the application continues operation, thus, the failure of $\mathbf{F_p}$ is masked from the client.

7.3 Replica Selection for Performance

Our final example illustrates WS-Naming as a means of selecting the "best" replica of a file at runtime based on the location of the client. By "best" we mean the "closest" in terms of bandwidth and latency. We will assume an existing mechanism that given a pair of network IP addresses returns the predicted latency and bandwidth between those two points. An example of such a service is the Network Weather Service (NWS)[25]. We will also assume a function "quality" that given the results from the NWS returns a positive integer – the higher the number, the "better" the connection.

There are several ways for this to work. The most simple is if the EPR for **F** initially contains a *wsa:Address* field that will cause an immediate *bad address* fault or can be identified by the client stub code, e.g., "NONE:". When the fault occurs, the client stub in **C** checks for a RR-EPR **S**. Assuming there is an RR-EPR **S**, the client calls **S** and asks for a new EPR for **F**.

This is where it gets interesting. On receiving the resolution request from **C**, **S** computes all pair-wise performance metrics between **C** and each of the replicas of **F**, **F[1..N]**. **S** selects the replica *k* that has the highest "quality" connection between **C** and **F[k]**. **S** then returns the EPR of **F[k]** to **C**, and processing continues as in the two earlier examples.

The three above examples are not exhaustive of all of the ways that WS-Naming can be used to support supporting SLAs. One can easily think of all kinds of ways to support high-availability or better performance via migration or replication. The point is that the mechanism is sufficient and simple.

8 Implementation

Specifications without implementations are always suspect. Is there some hidden flaw? Is the specification implementable with reasonable complexity and effort? How is the performance? Because these questions need to be answered the OGF requires two separate implementations of a proposed specification or profile before promoting the specification to full recommendation.

The Genesis II [14] team at the University of Virginia has implemented WS-Naming. Genesis II is an open source Grid middleware that is being used as a testbed or a number of specifications. Genesis II is implemented in Java and implements both aspects of WS-Naming, EPIs and resolvers. All Genesis II EPRs contain an EPI. Our replicated ByteIO [13] service embeds resolver EPRs into ByteIO resource endpoint EPRs.

Genesis II clients are WS-Naming aware and use the techniques described earlier to re-bind stale and invalidated WS-Addressing endpoints. Genesis II clients also use EPIs as cache handles for storing the results of cacheable invocations. In Genesis II a client-side Java proxy object is used to communicate with remote Web Service endpoints. This proxy object acts as a central manager for much of the grid client/server interaction details such as SOAP header manipulation (for WS-Addressing, security, etc.) and SOAP attachment management. It also provides a single point where WS-Naming rebinding takes place. Inside this stub, code around the remote invocation watches for failures that would indicate a communication problem (wrong address, wrong security information, etc.).

Any fault condition that it interprets as resulting from stale binding information causes this code to enact the re-resolution process describe in the preceding paragraphs.

High performance and Web Services are not often associated with one another. There is extensive protocol processing going up and down the Web Services stack that results in significant RPC times even for simple calls. When you add in SSL, authorization checking, and state management via a database, the overhead adds up.

That said, we implemented a simple test to determine the overhead of rebinding in our environment. The equipment is a 1.8 GHZ dual-core Intel running Windows Vista. We instantiated a ByteIO service in one container and replicated it another. A resolver service was located in the same container as the second copy. We then "warmed up" the Java environment – causing the JIT to compile and load classes etc. Next, we called the "ping" operation on the ByteIO endpoint and measured the round-trip time to the primary copy.

We then destroyed the primary copy container, and timed the "ping" operation again. This "re-bind" time below includes the time to detect that the primary no longer exists, invoke the resolver to get a new binding, and call the secondary.

Operation	Time with security	Time without security
Call primary	249	34 mS
Re-bind (call/fail primary, resolve, call secondary)	1572	1123 mS

Note that the full security stack was in place with both transport and message level encryption, strong mutual authentication, and full access control checks (including verifying X.509 certificates).

9 Summary and Future Work

Migration, replication, and failover are techniques that have been used for decades to realize service level terms such as performance, availability, reliability, and cost.

Similarly, endpoint abstraction, and in particular endpoint transparency via multi-level naming schemes, has been used in the distributed systems community for decades. The benefit to application developers is that they can focus on the application logic, and leave implementation of SLTs to lower layer software.

Web Services are becoming the standard means of service interaction in large scale distributed systems, service oriented architectures, and Grids. WS-Addressing EPRs are the standard mechanism to address Web Service endpoints. WS-Naming extends WS-Addressing by profiling the use of WS-Addressing extensibility points to include both unique Endpoint Identifiers and the ability to re-bind EPRs using a Resolver porttype.

The ability to rebind EPRs is a powerful tool. In this paper, we demonstrate how WS-Naming can be used in the implementation of four use cases: migration closer to a heavy user to meet performance SLTs, migration away from a failing or overloaded resource to meet performance, availability and reliability SLTs, recovery from a failed resource to meet reliability and availability SLTs, and replica management to meet performance SLTs. These examples are just the beginning and all sorts of algorithms and semantics can be implemented.

To "test out the specification" we have implemented WS-Naming in the Genesis II project. In that context we implemented service failover for replicated read-only ByteIO services. We are in the process of implementing a complete endpoint migration service, and primary copy replication service that will be used initially for ByteIO and RNS[8] [16] services. When WS-Naming rebinding is combined with smart stubs as in our Java implementation, applications are completely insulated from the rebinding process and the redirection of communication that ensues.

Acknowledgements: Marvin Theimer (Microsoft), Dave Snelling (Fujitsu), Frank Siebenlist (ANL), Hiro Kishimoto (Fujitsu), Andreas Savva (Fujitsu), Bill Horn (IBM), Mike Behrens (R2AD LLC), Alan Luniewski (IBM), and Jay Unger (IBM).

This material is based upon work partially supported by the National Science Foundation under Grant No. 0426972. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

[8] RNS or Resource Namespace Service is a Web Services specification for naming web resources in a hierarchical, human-readable way reminiscent of directories or folders in modern operating systems.

References

1. ByteIO Specification, Global Grid Forum. GFD-1.046, 20 Jun. 2005.
2. The Handle Technical Manual, CNRI, 2000.
3. Life Science Identifiers Specification v 1.0, Object Management Group, 2004.
4. Box, D., Christensen, E., Curbera, F., Ferguson, D., Frey, J., Hadley, M., Kaler, C., Langworthy, D., Leymann, F., Lovering, B., Lucco, S., Millet, S., Mukhi, N., Nottingham, M., Orchard, D., Shewchuk, J., Sindambiwe, E., Storey, T., Weerawarana, S. and Winkler, S. Web Services Addressing (WS-Addressing), W3C, 2004.
5. Cheriton, D. The V Distributed System. *Communications of the ACM*, *31* (3). 314-333.
6. Chervenak, A., Foster, I., Kesselman, C., Salisbury, C. and Tuecke, S. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets. *Journal of Network and Compute Applications*, *23*. 187-200.
7. GGF. WS-Naming Specification, Global Grid Forum, GFD-WS-Naming WG, http://forge.gridforum.org/projects/ws-naming-wg, 10 August 2005.
8. Grimshaw, A.N.-T.a.A.S. Using Reflection for Incorporating Fault-Tolerance Techniques into Distributed Applications. *Parallel Processing Letters*, *9* (2). 291-301.
9. Gudgin, M., Hadley, M. and Rogers, T. Web Services Addressing 1.0 – Core, World Wide Web Consortium, 2006.
10. Huang, Y., Kintala, C., Kolettis, N. and Fulton, N.D., Software Rejuvenation: Analysis, Module and Applications. in *25th Symposium on Fault Tolerant Computing, FTCS-25*, (Pasadena, California, 1995), IEEE, 381–390.
11. Jul, E., Levy, H., Hutchinson, N. and Black, A. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, *6* (1). 109-133.
12. Lee, I., Iyer, R.K. and Tang, D., Error/Failure Analysis Using Event Logs from Fault Tolerant Systems. in *Proceedings 21st Intl. Symposium on Fault-Tolerant Computing*, (1991), 10-17.
13. Morgan, M. ByteIO Specification 1.0, Global Grid Forum, 2005.
14. Morgan, M. and Grimshaw, A., Genesis II - Standards Based Grid Computing. in *Seventh IEEE International Symposium on Cluster Computing and the Grid*, (Rio de Janario, Brazil, 2007), IEEE Computer Society, 611-618.
15. Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M. and Welch, B. The Sprite Network Operating System. *IEEE Computer*, *21* (2). 23-36.
16. Pereira, M., Tatebe, O., Luan, L., Anderson, T. and Xu, J. Resource Name Service Specification.

17. Pike, R., Presotto, D., Thompson, K. and H.Trickey, Plan 9 from Bell Labs. in *UKUUG Summer 1990 Conference*, (1990).

18. Powell, M.L. and Miller, B.P., Process Migration in DEMOS/MP. in *9th ACM Symposium on Operating System Prinicples*, (1983), 110-119.

19. Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Lonard, P. and Neuhauser, W., Overview of the Chorus Distributed Operating System. in *USENIX Workshop on Micro-kernels and Other Kernel Architectures*, (1992), USENIX, 39-70.

20. Saltzer, J., Reed, D. and Clark, D. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems, 2* (4). 195-206.

21. Satyanarayanan, M. Scalable, Secure, and Highly Available Distributed File Access. *IEEE Computer, 23* (5). 9-21.

22. Tanenbaum, A.S., Renesse, R.v., Staveren, H.v., Sharp, G.J., Mullender, S.J., A.J., J. and Rossum, G.v. Experiences with the Amoeba Distributed Operating System. *Commuications of the ACM, 33* (12). 46-63.

23. Tannenbaum, A.S. and Steen, M.V. *Distributed Systems: Principles and Paradigms*. Prentice Hall, Upper Saddle River, New Jersey, 2002.

24. Tewari, V. and Milenkovic, M. Standards for Autonomic Computing. *Intel Technology Journal, 10* (4).

25. Wolski, R. Dynamically forecasting network performance using the Network Weather Service. *Cluster Computing, 1* (1). 119-132.