A Peer-To-Peer Infrastructure for Resilient Web Services

Stuart J. Norcross, Alan Dearle, Graham N.C. Kirby and Scott M. Walker
School of Computer Science, University of St Andrews, St Andrews, Fife KY16 9SS, Scotland
{stuart, al, graham, scott}@dcs.st-and.ac.uk

Abstract

This paper describes an infrastructure for the deployment and use of Web Services that are resilient to the failure of the nodes that host those services. The infrastructure presents a single interface that provides mechanisms for users to publish services and to find the services that are hosted. The infrastructure supports the autonomic deployment of services and the brokerage of hosts on which services may be deployed. Once deployed, services are autonomically managed in a number of aspects including load balancing, availability, failure detection and recovery, and lifetime management. Services are published and deployed with associated metadata describing the service type. This same metadata may be used subsequently by interested parties to discover services.

The infrastructure uses peer-to-peer (P2P) overlay technologies to abstract over the underlying network to deploy and locate instances of those services. It takes advantage of the P2P network to replicate directory services used to locate service instances (for using a service), Service Hosts (for deployment of services) and Autonomic Managers which manage the deployed services. The P2P overlay network is itself constructed using novel middleware based on Web Services and a variation of the Chord P2P protocol, which is self-configuring and self-repairing.

1. Introduction

Increasingly, e-services are deployed using Web Services mechanisms [1-3]. In many situations deployment onto a single physical server node would yield unacceptable levels of availability or latency, since the centralisation of the e-service would leave it vulnerable to failure or over-loading of the single node. This problem can be addressed by replicating the e-service on multiple physical nodes. If a particular node then fails, acceptable service may still be provided by the remaining replicas.

Clearly this introduces problems of coherency management for stateful services. Even with state-less services, however, there are several other difficulties with the approach. One is the need to predict in advance the maximum load and the frequency of node failure, so that appropriate levels of provisioning can be chosen. Given that loads on internet services are notoriously bursty [4], it is difficult to select a level of provisioning that will give high availability at reasonable cost. This is because maximum loads are orders of magnitude higher than average loads. Another problem is that although the e-service itself may be replicated, clients need to be able to locate a specific instance. If the directory that supports this look-up is itself centralised, then the e-service may become unreachable if the directory node fails or becomes overloaded.

The latter problem may be addressed by replicating the access directory, introducing in turn the need to manage coherency as e-services are added and removed.

The issue of retaining flexibility in provisioning levels can only be addressed by allowing dynamic adjustment of replication levels in response to failures or changes in access patterns. This applies equally to nodes hosting e-services and to nodes hosting elements of the directory. The management of this dynamic adjustment may be designed as an autonomic process [5], which monitors the state of the system, detects significant deviations from the desired state, and carries out actions to restore equilibrium. Such autonomic management addresses the problem that any statically configured distributed system is likely to decay over time, as failures accumulate and access patterns change. Given its importance, the autonomic process should also be replicated to avoid it becoming a point of failure, giving rise to the need for autonomic management of the autonomic management process itself.

This paper describes an infrastructure designed to support the requirements outlined above. It allows a service provider to deploy resilient e-services as Web Services without regard to the details of placement and replication factors. The e-services are automatically deployed onto an appropriate number of server nodes, and the number of these replicas is actively managed by the infrastructure in response to failures and changes in access patterns. The infrastructure provides a simple interface through which users may locate particular e-service instances.

The infrastructure separates provision of e-services from the provision of physical server hosts, thus supporting distinct markets in the two. An e-service provider may

buy server provision from any number of server providers, and have a particular e-service spread across them. To control this, the autonomic management layer provides interfaces for policy to be specified by both e-service and server providers.

The implementation of the infrastructure exploits work by others on peer-to-peer (P2P) overlays, and a generic resilient storage data structure that is applied at a number of different levels.

Section 2 discusses related work and describes some enabling technologies used in the infrastructure: RAFDA Run Time (RRT), Cingal and JChord. Section 3 describes the individual constituent components of the infrastructure, describing the Service Directory for locating service instances, the Host Directory for locating Service Hosts, and finally the autonomic mechanisms for the management of deployed services.

2. Related work

2.1 Web services

Web Services provide a remotely accessible service to programs executing in different address spaces in the form of a set of remotely accessible methods. Using typical Web Service technologies such as Apache Axis [2] and Microsoft .NET Web Services [3], services reside within execution environments that instantiate the service objects implementing their functionality, manage their lifetimes and handle requests to the contained services.

A web service is implemented by creating a class that conforms to the interface specified by the web service. The service is deployed by specifying its name, publicly accessible methods and the class that implements it. During a Web Service invocation, the container performs the actual call on the underlying service object. The container creates this service object by instantiating the implementation class using a statically defined constructor. It is not possible to specify a particular object to act as the service object, thus removing fine-grained control over the services from the programmer.

The deployment of a web service makes a single instance of the web service available via the container. To deploy the web service in multiple containers, the programmer must perform a deployment operation on each container. Existing Web Service systems do not permit the replication of a service across multiple containers.

Once deployed, the service is accessible only via its container and it cannot be migrated from one container to another. In the event of container or host failure, or under high-load, the web service may become unavailable. The deployment of Web Services in a resilient manner is difficult.

Universal Description, Discovery and Integration (UDDI) [6] is a specification for the creation of distributed registries of Web Services. It provides a mechanism to publish and discover information about businesses, the kinds of services they provide and how these services may be accessed. When creating a program that wishes to make use of a particular kind of service, the programmer searches a UDDI registry to obtain a suitable service based on business name, keywords or predefined taxonomies.

Part of the information returned from the UDDI registry is metadata describing how to bind to the service. Using this information, the programmer creates an application that accesses the service. Each service has a unique ID and so if it fails or is to be relocated, the service provider can indicate this fact in the UDDI registry. A program that attempts to access the service at an invalid location can interrogate the UDDI registry to obtain the new service location and transparently retry the call.

UDDI provides a distributed Web Service look-up mechanism and can also indicate to service users that a particular Web Service has been relocated. It does not provide tools for the autonomic deployment or replication of the services themselves and, by retaining a one-to-one mapping between service identifiers and service instances, does not support the look-up of replicated services.

2.2 ServiceGlobe

The system that is most closely related to the one described in this paper is ServiceGlobe [7]. It supports the storage, publishing, deployment and discovery of e-services. The system supports a variety of services; however, those of most interest are the so-called *dynamic services*, which can be deployed and executed on arbitrary ServiceGlobe servers. Services are discovered using UDDI, which yields the address of a Service Host. Services are implemented by mobile code (written in pure Java) that resides in code repositories. When a service request arrives at a Service Host the implementing code is fetched, if necessary, from the code repositories and executed on a runtime engine residing on the host. The major differences between the technology described in this paper and the ServiceGlobe technology is that ServiceGlobe appears to be aimed at transient stateless services.

2.3 RAFDA Run Time (RRT)

We now introduce several of the building blocks of our infrastructure: RRT, Cingal and JChord.

The RRT [8] has been developed as part of the RAFDA project (Reflective Application Framework for

Distributed Architectures) [9]. The RRT permits arbitrary components from arbitrary applications to be exposed for remote access. It allows the dynamic deployment, as Web Services, of components in a running application. Remote method calls can be performed on the exposed components from different address spaces, possibly on different machines. The RRT has five notable features that differentiate it from typical Web Service technologies:

1. Specific, existing component instances rather than component classes are deployed as Web Services.
2. The programmer does not need to decide statically which component classes support remote access. Any component from any application, including previously compiled applications, can be deployed as a Web Service without the need to access or alter the application's source code.
3. A remote reference scheme, synergistic with standard Web Services infrastructure, provides pass-by-reference semantics, in addition to the pass-by-value semantics supported by Web Services.
4. Parameter passing mechanisms are flexible and may be dynamically controlled through policies. A deployed component can be called using either pass-by-reference or pass-by-value semantics on a per-call basis.
5. The system automatically deploys referenced components on demand.

2.4 Cingal

The Cingal system [10-12] (Computation IN Geographically Appropriate Locations) supports the deployment of distributed applications in geographically appropriate locations. It provides mechanisms to execute and install components, in the form of *bundles*, on remote machines. A bundle is the only entity that may be executed in Cingal, and consists of a signed XML-encoded closure of code and data and a set of bindings naming the data. Cingal-enabled hosts contain appropriate security mechanisms to ensure malicious parties cannot deploy and execute harmful agents, and to ensure that deployed components do not interfere with each other either accidentally or maliciously. Cingal components may be written using standard programming languages and programming models. When a bundle is received by a Cingal-enabled host, provided that the bundle has passed a number of checks, the bundle is *fired*, that is, it is executed in a security domain (called a *machine*) within a new operating system process. Unlike processes running on traditional operating systems, bundles have a limited interface to their local environment. The repertoire of interactions with the host environment is limited to: interactions with a local store, the manipulation of bindings, the firing of other bundles, and interactions with other Cingal processes.

The Cingal infrastructure includes a number of services that may be invoked from bundles executing within machines. These are the *store*, *store binder*, *process binder* and *Valid Entity Repository* (VER) providing storage, binding and certificate storage respectively. Cingal implements a two-level protection system. The first level of security restriction is on the firing of bundles. A conventional Unix or Windows style security model is not appropriate for Cingal hosts, which do not have users in the conventional sense. Instead, security is achieved by means of digital signatures and certificates. Each host maintains a list of trusted *entities*, each associated with a security certificate. Entities might correspond to organisations, humans or other Cingal hosts. This data structure is maintained by the VER. Bundles presented for firing from outwith a Cingal host must be signed by a valid entity stored in the VER. The VER maintains an associative data structure indexed by the entity *id* and mapping to a tuple including certificates and rights. Operations are provided for adding and removing entities from the repository. Of course these operations are subject to the second protection mechanism, which is capability based.

2.5 JChord

JChord is our implementation of the Chord [13] peer-to-peer look-up protocol. This implementation provides a peer-to-peer overlay that supports Key-Based-Routing (KBR) [14] for addressing nodes in the underlying network. Under a KBR scheme every entity addressable by an application has an associated *m*-bit key value (where *m* is a system constant), and every key value maps to a unique live node in the overlay network. Up-calls from the routing layer inform the application layers of changes to the key space, thus allowing an application to be aware of changes to the set of keys that map to the local node.

Chord is a ring-based protocol, which at the simplest level requires each node to maintain only a pointer to its immediate successor in the ring. Each node also has a unique key and the ring is arranged in key order modulo 2^m. The Chord protocol supports a single *lookup* operation, which takes a key value and returns the network address of the Chord node to which the key value maps. A look-up on key K will yield the address of the node N whose key K_N is the first of the ring members to succeed K in the key space. In this way the Chord protocol provides a distributed hash function that maps from keys to overlay nodes. Each node maintains a list of nodes that follow it in the ring, known as its *successor list*. A suc-

cessor list of size *l* allows the ring to survive the failure of up to *l-1* adjacent nodes. This provides resiliency of the ring and the look-up protocol, though further measures are required to ensure integrity of the data structures hosted by ring nodes.

The JChord implementation consists of a set of Java classes that provide the base Chord functionality, used by higher-level applications in constructing a key-addressable peer-to-peer overlay network. JChord uses the RRT to publish two interfaces that facilitate inter-node communications. The first of these interfaces (which is not discussed here) supports the core Chord functions used in maintaining the ring. The second interface, *JChordAPI*, provides the *lookup* method as shown in Figure 1.

```
public interface JChordAPI {
  URL lookup(Key k)
}
```

Figure 1. JChord interface for key look-up

The RRT allows arbitrary objects within a JVM to be exposed to the network as Web Services. Using this mechanism, individual JChord nodes may expose any number of additional services. In the system described in this paper, each JChord node hosts a number of additional services including a Service Directory that supports the discovery of Web Services, a Host Directory supporting the discovery of machines willing to host services, and a Data Storage service.

These services utilise a generic key-based data storage architecture similar to the DHash layer used in the Cooperative File System (CFS) [15]. Data is stored on a *root* node and replicated (for fault tolerance) on a number of successive nodes in the ring. The generic storage layer on each JChord node is exported using a Data Storage service interface (not elaborated here). This service is also used to implement other services exported by each JChord node as described above.

Data *D* with key *K* is stored at the JChord node *N* that is returned by JChord's *lookup* method when called with *K*. *N* is known as the *primary node* for *D* and the copy of *D* stored at *N* is called the *primary copy*. When data is first stored at a primary node, that node is responsible for ensuring that the data is replicated for fault tolerance. This is achieved by storing the data on the next *r* nodes in the ring, where *r* is some autonomically managed aspect of the system which may be equal to *l*, the successor list length, but does not have to be.

Changes to the ring's membership impact on the data storage layer. For example, a new node inserted into the ring may become the primary node for data already stored on another node. Consequently, following such an insertion, some existing data must be copied onto the new primary node. Similarly, data may have to be replicated further as nodes holding replicas of data fail or leave the ring. Thus the data storage layer needs to be aware of changes in the ring topology. To accommodate this, JChord provides an up-call mechanism, which informs the generic storage system and other high-level components of changes in the ring topology.

3. An Infrastructure for Resilient Web Services

The primary concern of this paper is an infrastructure for the deployment and use of Web Services that are resilient to the failure of the nodes that host those services. The external interface to this infrastructure is shown in Figure 2.

```
public interface ResilientWSInfrastructure {
  URL lookupService(URI service_id);
  void publishService(URI service_id,
        URL location);
  void addServiceHost(Certificate[] certs,
        URL service_host);
  void deploy(Bundle impl, URI service_id
        DeploymentSpecification spec);
  URL lookupManager(URI service_id);
  void registerManager(URL am,
        URI service_id);
}
```

Figure 2. Resilient infrastructure interface

The two simplest methods provided by this interface are *lookupService* and *publishService*. The *publishService* method is used by a service provider to inform the system of the URL of a new service instance. It has two parameters: a URI used to uniquely identify the service, and a URL denoting a deployment of that service. Although we assume that a user wishing to locate an instance of a service is in possession of the service URI, we are planning to add a fuzzy matching service to complement this whereby a user can discover service URIs from partial information or metadata such as keywords, WSDL fragments etc. An *unpublishService* method, omitted for brevity, takes the same parameters and is used to remove information about a published service from the system. A client wishing to make use of a service passes its URI to the *lookupService* method, and receives the URL of an instance of the service.

The implementation of the *ResilientWSInfrastructure* interface utilises a number of services residing on the JChord ring. All services are exported by every JChord node, and are made resilient by replicating their state on the ring as described in the previous section.

The first of these services is the Service Directory which implements the *IServiceDirectory* interface shown in Figure 3. It provides a mechanism to resolve a key for a service, derived from its URI, to the network address of an instance of that service—by implementing a one-to-many mapping from each service key to the URLs of the instances of that service.

Within the infrastructure, Web Services are identified by keys. Given a service *S*, its key K_S is obtained by applying a globally known hash function *H* to the URI for *S*, URI_S. Thus:

$$K_S = H(URI_S)$$

```
public interface IServiceDirectory {
  void store(Key key, URL location);
  void remove(Key key, URL location);
  URL[] findAll(Key key);
  URL findOne(Key key);
}
```

Figure 3. Service Directory interface

The Service Directory instance located on a JChord node *N* is responsible for storing the location information for instances of all Web Services whose keys map to *N* under the JChord look-up protocol.

To find the location of a node hosting service *S* with key K_S, a JChord look-up is executed on K_S to yield the JChord node *N* hosting the Service Directory instance that contains the entries for K_S. The Service Directory at *N* is then queried for the locations of nodes hosting service *S*. The same look-up process is executed when publishing a new location for a service, although here the final step is to call to the Service Directory to add a new location entry for K_S.

The RRT on each JChord node exports an *IServiceDirectory* interface as a web service bound to the name "ServiceDirectory". Like all services deployed using the RRT, a reference to a Service Directory may be obtained by specifying the URL of the remote RRT and the service name with which that service is associated on that host. This is provided by the RRT's *getServiceByName* method.

The *lookupService* and *publishService* methods from the implementation of the *ResilientWSInfrastructure* interface can now be expanded as shown in Figure 4. The *unpublishService* method is omitted; however, the implementation is similar to *publishService*.

```
URL lookupService(URI id) {
  Key k = H(id);
  URL sdLocation = JChord.lookup(k);
  IServiceDirectory sd = (IServiceDirectory)
     RRT.getServiceByName(sdLocation,
           "ServiceDirectory");
  return sd.findOne(k);
}

void publishService(URI id, URL location) {
  Key k = H(id);
  URL sdLocation = JChord.lookup(k);
  IServiceDirectory sd = (IServiceDirectory)
     RRT.getServiceByName(sdLocation,
           "ServiceDirectory");
  sd.store(k, location)
}
```

Figure 4. lookupService and publishService

An example of the system running is shown in Figure 5. The JChord ring is represented by the five nodes arranged in a pentagon in the centre of the diagram. Each node exports a Service Directory implementation, some of which are shown simply by the interface, and some of which are elaborated to show their contents. A number of Web Services are arranged around the outside of the diagram offering services identified by the URIs S_X, S_Y and S_Z. The hash of S_X maps to node *R2* in the ring, whereas the hashes of S_Y and S_Z map to node *R4*. Thus a look-up of the URI S_Y from any location will map to node *R4*, where an instance of *IServiceDirectory* can be found and its *findOne* method invoked to yield an instance of the S_Y service.

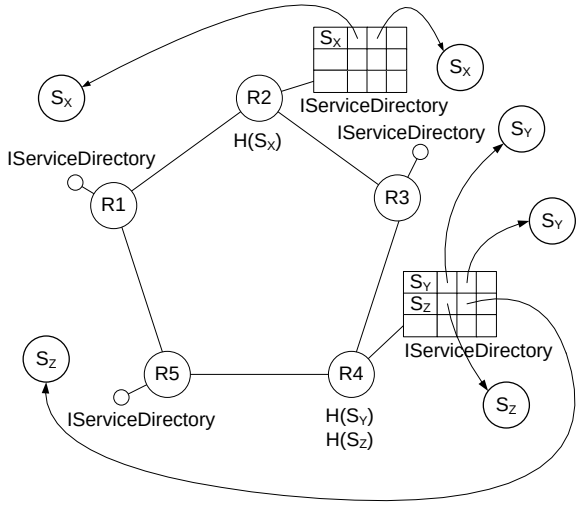

Figure 5. Example system state

3.1 Deploying services

Thus far, we have shown how service providers can advertise extant services and users may locate them. The infrastructure supports the autonomic deployment of services and the brokerage of hosts on which services may be deployed. This section deals with issues surrounding service deployment.

In order to support automatic deployment, three additional elements of functionality are required: the ability for hosts known as Service Hosts to advertise their willingness to host services, the ability to supply the service code implementing services, and the ability to deploy the service code on available hosts. In the *ResilientWSInfrastructure* interface, the first of these functions is provided by the *addServiceHost* method while the others are provided by the *deploy* method.

The *addServiceHost* method allows Service Hosts to be registered with the infrastructure. It is parameterised with the URL of a host capable of hosting services and an array of certificates, one for each entity for whom the Service Host is prepared to host a service. The certificates

are used to identify individuals or organisations that wish to deploy services onto that host.

This method utilises another service, the Host Directory, which implements the *IHostDirectory* interface shown in Figure 6.

The algorithm to locate the appropriate Host Directory is similar to that used to locate the appropriate Service Directory. However, in the case of Host Directories, the hash function is based on the certificate passed to the *addServiceHost* and *removeServiceHost* methods (again not shown) rather than on a URI.

```
public interface IHostDirectory {
  void store(Key key, URL serviceHost);
  void remove(Key key, URL serviceHost);
  URL[] findAll(Key key);
  URL findOne(Key key);
}
```

Figure 6. Host Directory interface

We assume a model where each service provision entity has an associated certificate and where each Service Host specifies a set of certificates for which it will host services. Thus a Service Host may be recorded in many different Host Directories, one for each of the certificates for which it will host services.

Thus a Service Host *S* registers itself with the infrastructure by creating a Host Directory entry for each certificate *C* for which *S* will host services using *addServiceHost*. A key K_C is generated for each certificate *C* by applying the globally known hash function *H* to *C*:

$$K_C = H(\ C\ )$$

The implementations of *addServiceHost* and *removeServiceHost* perform JChord look-ups on K_C, yielding the Host Directory on which the appropriate *store* or *remove* operations may be invoked.

The *deploy* method, shown in Figure 7, in the *ResilientWSInfrastructure* interface, permits a service provider to deploy a service onto a set of Service Hosts. The *deploy* method takes three parameters: a Cingal bundle containing the code and data implementing the service and signed by an appropriate entity; a deployment specification (not elaborated here) which includes information such as the number of instances to be deployed; and the URI identifying the service.

The implementation of *deploy* first extracts the certificate *C* from the bundle; this certificate is hashed to obtain the key K_C. Next, it performs a JChord look-up on K_C to yield the appropriate Host Directory, and calls the *findAll* method which returns all of the Service Hosts prepared to host services signed with certificate *C*. Based on the deployment specification an appropriate subset of these Service Hosts is chosen. Next, the bundle is fired on each one, making the service available at that host. This operation will normally succeed, but is subject to a certificate check performed by the Service Host. Finally, each of the newly deployed bundles is registered with the infrastructure using the *publishService* method provided by the *ResilientWSInfrastructure* interface.

```
void deploy(Bundle b,
       DeploymentSpecification spec, URI u) {
  Certificate c = b.getCertificate();
  Key k_cert = H(c);
  URL hdLocation = JChord.lookup(k_cert);
  IHostDirectory hd = (IHostDirectory)
       RRT.getServiceByName(hdLocation,
       "HostDirectory");
  URL[] host_urls = hd.findAll(k_cert);
  //Choose hosts based on
  //DeploymentSpecification parameter.
  URL[] chosen_urls =
       chooseHosts(spec, host_urls);
  for ( URL i : chosen_urls) {
     Cingal.fire(bundle, i);
     RWS_Infrastructure.publishService(u,i);
  }
}
```

Figure 7. The deploy method

Figure 8 shows six service instances deployed on five nodes; five of these services are hosted by Service Hosts. Service Hosts *A*, *B* and *D* are prepared to accept bundles signed with certificate *Cert1*. *H*(*Cert1*) maps to *R1* and consequently, the URLs for these Service Hosts are held in the Host Directory at ring node *R1*. Similarly, the URLs of the Service Hosts willing to host services signed with certificate *Cert2* are held at *R4*. From the diagram it can be seen that the deployer of service S_X holds certificate *Cert1* whereas the deployer of service S_Y holds certificate *Cert2*. One instance of the service S_Z is not hosted by a Service Host and has been independently instantiated and its URL published. The URLs of the service instances are stored as in Figure 5.

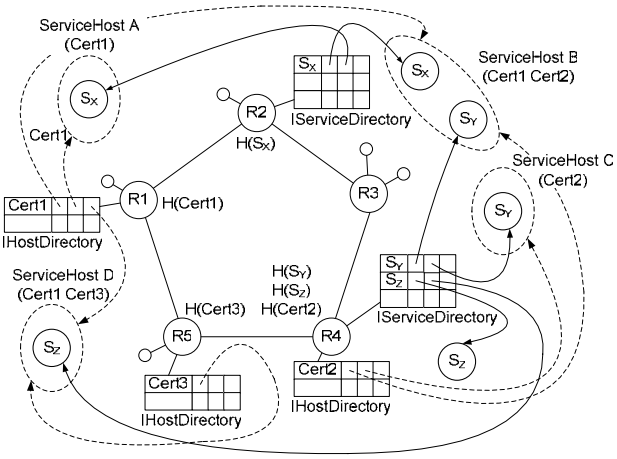

Figure 8. Service Hosts and service instances

3.2 Autonomic services

The infrastructure described thus far provides the functionality to deploy and locate instances of Web Services. The infrastructure on which it is implemented is resilient to failure. However, the availability of the repli-

cated services is likely to decay over time due to failures, etc. To address this problem, a framework that supports autonomic management is required. Autonomic Managers are capable of deploying new service instances to replace those that have failed and to stop or deploy services in response to load changes. The long-term, reliable storage and retrieval of service implementation bundles is necessary for lifetime management of services by Autonomic Managers. To be able to deploy a service, an Autonomic Manager needs to be able to access a copy of the bundle used to originally instantiate the service. This requirement is addressed by the Data Storage service provided on each node of the JChord ring.

A number of parties could be responsible for storing the bundle in the Data Storage service. However, the obvious place for the activity to be carried out is in the *deploy* method. The mechanism for storing bundle data in the Data Storage service is similar to the other ring activities described in this paper and is consequently omitted.

In order to instantiate new services and stop undersubscribed services, the Autonomic Manager must have the same credentials as the entity that instantiated the service. If this were not the case, any arbitrary process could pose as an Autonomic Manager and maliciously start and stop services. Consequently, the Autonomic Manager must be signed with the same certificate as the supplier of the service being managed. A corollary of this is that Autonomic Managers must themselves be self-managing since they cannot be safely managed by a third party.

As described above, Autonomic Managers are self-managing. In order for them to be resilient to failure, a service provider must instantiate multiple manager processes for any given service, and the redundant replicas peered together. Thus, for some particular service, a set of Autonomic Managers exists. Whilst this might suggest a multiplicity of managers, a set of managers may manage more than one service.

Services find their Autonomic Managers using the *lookupService* method described earlier. Similarly, this same mechanism allows an Autonomic Manager to find its peers. However, for efficiency, Autonomic Managers will normally use some optimised method of inter-peer communication, such as a ring-based protocol, in order to maintain coherent state.

Managers must be capable of instantiating new copies of themselves in response to overload, peer failure etc, in the same manner as the instantiation of the services they manage. Clearly, to bootstrap this process, some managers must be manually deployed on some set of willing hosts.

Autonomic Managers must be compliant with the *IAutonomicManager* interface, shown in Figure 9. Users may write their own implementations; however, library code is provided that implements a standard Autonomic Manager. This code must be signed by the entity (organisation or user) providing the service and deployed either using the *deploy* method or some other technology. In order that instances of some service *S* may find their Autonomic Manager AM_S, Autonomic Managers register themselves using the *registerManager* method in the *ResilientWSInfrastructure* interface. Instances of service *S* can look up their Autonomic Manager using the *lookupManager* method which yields an instance of an Autonomic Manager. Service instances use the *report* method to report information such as service load etc. Based on the aggregate information gleaned from the service instances, the Autonomic Manager AM_S can start and stop instances of *S*.

```
public interface IAutonomicManager{
  void report( AutonomicData d );
}
```

Figure 9. Autonomic Manager interface

The implementations of *registerManager* and *lookupManager* are similar to those for *publishService* and *lookupService* and use the same Service Directory service. A service key $K_{S\text{-}AM}$, corresponding to the Autonomic Managers for a service *S*, is generated by hashing the concatenation of the service's URI U_S and a globally known salt string AM_{SALT}. The aim of the salt string is to avoid creating a bottleneck whereby a single JChord node is responsible both for instances of a service and the Autonomic Managers for that service. Thus:

$K_{S\text{-}AM} = H (U_S + AM_{SALT})$

A JChord look-up on $K_{S\text{-}AM}$ yields the Service Directory where the URLs of each of the Autonomic Managers for the service are stored. The methods *registerManager* and *unregisterManager* (again, omitted for brevity) call the *store* or *remove* methods (respectively) on the interface while *lookupManager* calls *findOne*. The same set of Autonomic Managers can be made responsible for the management of a number of different services by registering their URLs with the URIs for those services.

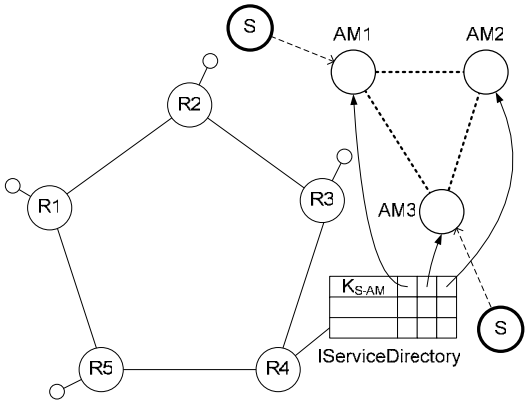

Figure 10. Peered Autonomic Managers

Figure 10 shows three peered Autonomic Managers *AM1*, *AM2* and *AM3*, all of which are responsible for

managing instances of service *S*. The key for the Autonomic Manager for service *S* maps to ring node *R4*. In the diagram it can be seen that two instances of *S* have obtained the URLs of two different Autonomic Managers.

4. Status and further work

The underlying components—RRT, Cingal and JChord—are fully implemented. We are currently working on a full implementation of the infrastructure, and aim to have a fully functional system by the time of the conference. We plan to carry out an experimental evaluation of the performance and resilience of the infrastructure, initially on a local cluster and then on PlanetLab.

We are also working on a web service discovery component, called the Discovery Service, that maps from some fuzzy meta-data to service keys. Examples of fuzzy data might include fragments of WSDL, the identity of providers, keywords etc.

5. Conclusions

We have presented the design of an infrastructure for resilient e-services. The resilience arises from dynamically managed replication of both the e-services themselves, and the directories used to locate instances of the e-services. Users and providers of e-services are shielded from most of the inherent complexity by the infrastructure's autonomic management capability, and the self-healing nature of the underlying P2P protocols.

Acknowledgements

This work was supported by EPSRC Grants GR/S44501 "Secure Location-Independent Autonomic Storage Architectures", GR/M78403 "Supporting Internet Computation in Arbitrary Geographical Locations" and GR/R51872 "Reflective Application Framework for Distributed Architectures", and by Nuffield Grant URB/01597/G "Peer-to-Peer Infrastructure for Autonomic Storage Architectures". Much of the JChord implementation work was carried out by Stephanie Anderson.

References

[1] W3C, "Web Services", 2004 *http://w3c.org/2002/ws/*

[2] The Apache Software Foundation, "Apache Axis", 2004 *http://ws.apache.org/axis/*

[3] Microsoft Corporation, "Web Services", *http://msdn.microsoft.com/webservices/*

[4] D. Clark, W. Lehr, and I. Liu, "Provisioning for Bursty Internet Traffic: Implications for Industry and Internet Structure", Proc. MIT ITC Workshop on Internet Quality of Service, 1999.

[5] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing", *IEEE Computer*, vol. 36 no. 1, pp. 41-50, 2003.

[6] UDDI.org, "UDDI Version 2.04 API, Published Specification", 2002 *http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.pdf*

[7] M. Keidl, S. Seltzsam, K. Stocker, and A. Kemper, "ServiceGlobe: Distributing E-Services Across the Internet", Proc. 28th International Conference on Very Large Databases (VLDB 2002), Hong Kong, China, 2002.

[8] Á. J. Rebón Portillo, S. Walker, G. N. C. Kirby, and A. Dearle, "A Reflective Approach to Providing Flexibility in Application Distribution", Proc. 2nd International Workshop on Reflective and Adaptive Middleware, ACM/IFIP/USENIX International Middleware Conference (Middleware 2003), Rio de Janeiro, Brazil, 2003.

[9] A. Dearle, G. N. C. Kirby, A. J. Rebón Portillo, and S. Walker, "Reflective Architecture for Distributed Applications (RAFDA)", 2003 *http://www-systems.dcs.st-and.ac.uk/rafda/*

[10] J. C. Diaz y Carballo, A. Dearle, and R. C. H. Connor, "Thin Servers - An Architecture to Support Arbitrary Placement of Computation in the Internet", Proc. 4th International Conference on Enterprise Information Systems (ICEIS 2002), Ciudad Real, Spain, 2002.

[11] *http://www-systems.dcs.st-and.ac.uk/cingal/*

[12] A. Dearle, G. N. C. Kirby, A. McCarthy, and J. C. Diaz y Carballo, "A Flexible and Secure Deployment Framework for Distributed Applications", in *Lecture Notes in Computer Science 3083*, W. Emmerich and A. L. Wolf, Eds.: Springer, 2004, pp. 219-233.

[13] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications", Proc. ACM SIGCOMM 2001, San Diego, CA, USA, 2001.

[14] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a Common API for Structured Peer-to-Peer Overlays", Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03), Berkeley, CA, USA, 2003.

[15] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-Area Cooperative Storage With CFS", Proc. 18th ACM Symposium on Operating Systems Principles, Banff, Canada, 2001.