Improving the Elasticity of Services with a P2P Infrastructure

Weidong Han and Ralph Deters

Department of Computer Science
University of Saskatchewan
Canada
{weh195, deters}@cs.usask.ca

Abstract. Web Services (WS) are one of the most popular approaches for building distributed systems due to their widespread acceptance within industry and academia, established and open standards, a wealth of development tools/infrastructures (e.g. Apache Axis [1]), and maybe most importantly the number of successful deployments. A key issue in the deployment of WS is the *elasticity* of services e.g. their ability to respond to changes in the demand by dynamically adding or removing WS provider nodes. This paper presents a novel P2P based management approach for WS providers, that allows organizations to dynamically add or remove replicated providers at runtime and thus increases their elasticity. Unlike the current centralized approaches, this scalable P2P approach allows for a fully distributed and fault-tolerant management of replicated provider and thus enables organizations to respond faster to changes in WS consumer behavior.

Keywords: SOA, Web Services, P2P, Gnutella, Elasticity.

1 Services & SOA

Within the context of service-oriented systems, services are computational elements that expose functionality in a platform-independent manner and can be described, published, discovered, orchestrated and consumed across language/ platform/organizational borders. Compared to other middleware, such as RPC (e.g. ONC-RPC) and object-oriented middleware (e.g. CORBA), service-orientation [2] differs in its lack of access and location transparency, since there is a very clear notion between local and remote. However, other transparencies namely migration, replication, concurrency, scalability, performance and failure transparencies can be supported in service-oriented middleware. Compared to objects, services are more autonomous, e.g. the can refuse invocation requests. And unlike objects that tend to be described in terms of data structures and methods, services are described by contracts and schemas and they tend to rely on policies to govern their behavior.

The idea of using services as the building block was popularized by the introduction of the Service-Oriented Architecture (SOA) [3] in 1996. Schulte and Natis [4] developed this architecture as a means to solve many of the enterprise integration and development challenges. In this classical SOA approach, three types of participants were identified, namely the service consumer, the service provider and a registry (e.g. UDDI Server). Service providers register their services with the registry which in turn is used by the consumers to locate services. The service registry acts as a service broker, which provides a public listing of registries which is exposed via a public API. SOA was designed to be a language and platform independent architecture and can consequently be implemented in a variety of ways (e.g. using CORBA).

SOAP based Web Services (WS) have emerged as the de facto standard for implementing SOA due to the well accepted and standardized WS* protocol stack. Three protocols form the core of WS* namely **S**imple **O**bject **A**ccess **P**rotocol (SOAP) [5], **W**eb **S**ervice **D**efinition **L**anguage (WSDL) [5] and **B**usiness **P**rocess **E**xecution **L**anguage (BPEL) [6]. Not surprisingly these three core WS* protocols are now supported by all virtually all mainstream programming languages allowing developers to easily expose application interfaces and/or consume existing services.

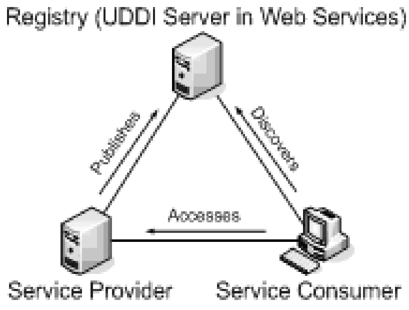

Fig. 1. Classical SOA

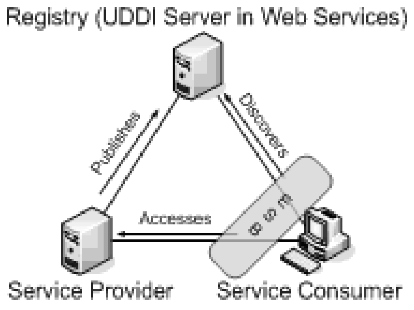

Fig. 2. SOA with ESB

However, as the development of providers and consumers was with the help of frameworks and IDEs simplified, it became apparent that the classical service-oriented architecture was to open for most businesses. Allowing consumers to discover the location and functionality of the physical providers and was seen as too risky. Therefore the classical architecture that provides consumers with direct access to the registry and allows them to directly engage providers was changed into an architecture that allows the organization/enterprise that owns the providers to control all aspects of discovery and service invocation. To provide an organization/enterprise with better control, SOA vendors offer an **E**nterprise **S**ervice **B**us (ESB) that controls the visibility of providers to external consumers, prevents direct consumer provider interaction and also enables fine grained control on when/how external consumers engage providers. The ESB acts like a proxy between consumers and providers controlling all communication between consumers and providers. Modern ESBs offer request/ response routing, mediation (e.g. service mapping, protocol transformation), orchestration (execution of workflows) and maybe most importantly, fine grained management thus allowing it to control who and how its providers are engaged.

Being able to control all aspects of how consumers discover and engage service providers has allowed businesses to create *virtual* service providers and to define how they are mapped to physical ones. By mapping one virtual provider to a set of replicated ones, it becomes possible to increase the capacity and fault-tolerance of services. If however customization is important (e.g. quality of service, QoS), the requests for the virtual providers are intercepted by the ESB, augmented and then re-routed to the physical one. By providing different augmentations and transformations of the consumer requests, it becomes possible to create different behaviors thus achieving customization.

2 Provider Overloads

An important aspect of using the WS* protocol stack is that the communication protocol SOAP uses XML messages, e.g. consumers and providers exchange XML documents over HTTP. Consumers invoke a service of a provider by using the HTTP POST command for the transmission of their SOAP request message. The providers respond to this POST command with a HTTP response that contains a SOAP response message. Since SOAP messages are XML documents, they tend to require more memory and processor resources to create and parse than other message formats.

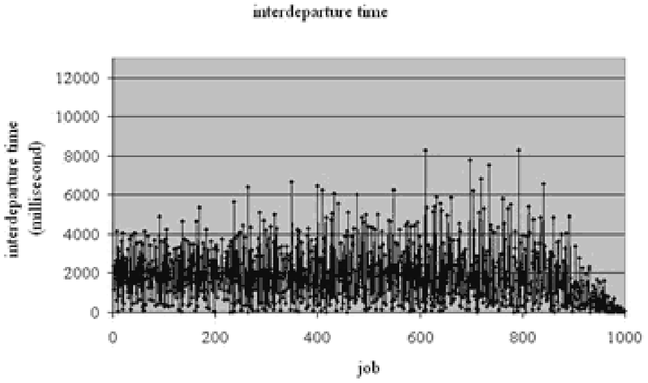

Fig. 3. Provider experiencing light Overload.

An example of this effect can be seen in figure 4, which show the inter-departure times of a slightly overloaded web service provider implemented in Java 6 (using the standard javax.jws packages). The service provider receives 1000 tasks at an arrival rate of 1.7 seconds/task. Since each task requires 100 % CPU for ca. 1.9 seconds the server experiences right away a slight overload. While it is expected that in an overload situations the departure times/rates fluctuate, it is interesting to note that the variance of the departure rates increases over time. Repeating the experiment with longer bursts shows that over time an increasingly chaotic behavior emerges and that if the bursts continue for too long the provider crashes due to memory errors. The main reasons for this chaotic behavior are the creation of too many threads (one for each request) and the XML parsing that result in the creation of many temporary objects.

Organizations therefore tend to avoid provider overloads and try to keep the utilization of their providers below 70 %. One of the most effective methods for preventing overloads within service-oriented systems is the use of an admission control. Since the ESB is already the main mechanism to control access to the providers it is the natural location to host an admission control. In its simplest form, the admission system is a queue. Incoming messages are always added to the queue and only if a provider is underutilized, a request is de-queued and sent to the provider. In case the provider is already fully utilized (e.g.

above 70% utilization), the request has to remain in the queue till a provider becomes available. Using an admission control also allows for the introduction of different levels of service. By moving from a simple to a priority queue and determining the priority of the request based on the sender it becomes possible to ensure that some customers receive better services by simply serving their requests sooner. Obviously, any organization that exposes its services will try to ensure a high availability and reliability of its services to stay competitive. This is achieved by replicating service providers. However, determining how many redundant providers should be used is very difficult when the usage of services is fluctuating. Since an idle X86 blade server consumes on average 65 % of the energy it would need during full utilization, it is important to avoid idle or severely underutilized machines - especially in light of growing environmental concerns and rising energy prices for cooling servers.

It becomes therefore important to dynamically add or remove physical providers in an effort to ensure optimal resource usage while meeting customer demands. The ability to dynamically scale up or down is referred to as elasticity. The more elastic a service is, the faster it can be scaled up/down. In organization/ enterprises that main their own physical machines, the elasticity of services is limited by the number of available machines. However, as cloud computing became more popular, organizations/enterprises were able to dramatically increase their capability by dynamically purchasing capacity from vendors like Amazon EC2 for peak demands.

3 Dynamic adding and removing providers

A key issue in the dynamic adding and removing of service providers is to ensure no interruption of service and a minimal overhead. As long as the changes are small and infrequent, a centralized solution e.g. use of the ESB is desirable. However, as the number of replicated providers is increased and the changes become more frequent a decentralized approach is needed to avoid a bottleneck and single point of failure. Consequently it is necessary to distribute the management of the redundant service providers. One fairly well tested mechanism for dealing with redundant and highly dynamic resources is a P2P network e.g. a Gnutella network.

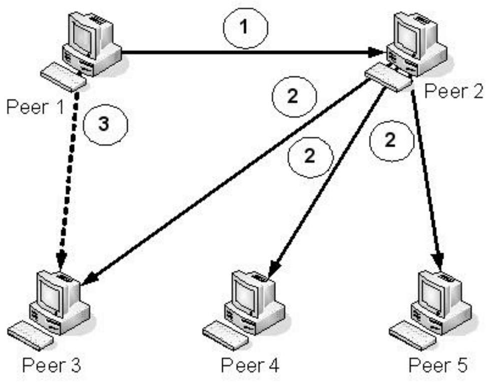

Fig. 4. Gnutella Peers.

A P2P network using the Gnutella protocol (0.4) consists only of peers that all provide common services and differ only in the resources they own. The peer providing a resource to others is the provider peer, while the peer consuming the resource is the consumer peer. All peers in a P2P network are organized by themselves using a specific protocol, by which they can publish and find resources in a cooperative pattern. Since P2P performs dynamic discovery to filter out unavailable resources, it is very robust in a dynamic networking environment. One of the most well studied and widely applied P2P protocols is Gnutella. Each Gnutella peer has four basic operations, Ping, Pong, Query, and QHit. Ping and Pong are used to find existing peers in the network, and Query and QHit are used to find desirable resources. A Gnutella peer should propagate an incoming request (Ping, Query) to those peers that it has direct connections with (those peers are usually called as its neighbors), and send back the response (Pong, QHit) to the peer issuing the request. Based on Figure 5, the working mechanism of Gnutella can be explained in the following four steps.

1. If Peer 1 wants to join a P2P network, it will first connect to a known peer, e.g., Peer 2 in Figure 5.
2. After a connection to Peer 2 is established, Peer 1 will send a Ping request to Peer 2 to find other peers. Peer 2 responds with a Pong message to Peer 1. The Ping request is also propagated to Peer 2's neighbors (e.g., Peer 3, 4, 5). All neighbors respond the Ping and send Pong messages back to Peer 1 through Peer 2. At this stage, Peer 1 knows Peer 2 – Peer 5 and vice versa.
3. Since a Gnutella peer always keeps a certain number of active connections (usually ≥ 5) to other peers, Peer 1 will try to establish more connections. For instance, Peer 1 may connect to Peer 3.
4. Peer 1 sends Peer 2 a Query request to find a desirable resource. In addition, Peer 2 propagates the Query request to its neighbors. If a peer has the requested resource, it sends a QHit message back to Peer 1 along the incoming path. In this way, Peer 1 knows all peers owning the requested resource. How Peer 1 accesses the resource of other peers depends on different implementations. For most systems, exchanging resources will use a dedicated connection instead of the one transferring requests.

The Ping-Pong mechanism of Gnutella keeps detecting any change in the peer while the Query-QHit mechanism ensures dynamic lookups.

4 Integrating the Gnutella protocol into the WS* Protocol stack

In Web Services, the SOAP message is the base of the inter-application communication. Its textual format provides an opportunity to manipulate the communication by using a proxy that intercepts SOAP messages passing through it. Figure 5 shows the concept of using a transparent proxy between a WS consumer and a WS provider, in which the proxy plays two roles. From the perspective of the consumer, the proxy is the service provider. From the perspective of the provider, the proxy is the consumer. Since the consumer sends the request to and receives the response from the proxy, the proxy can fully control the communication between the consumer and the provider. In Figure 5, the proxy can "select" a service provider dynamically without notifying the consumer.

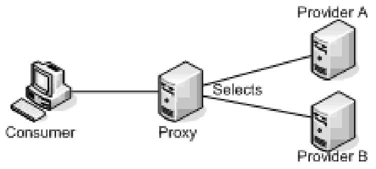

Fig. 5. Using a Proxy to manipulate communication.

The P2P Web Services Framework (PWSF), that seamlessly integrates P2P protocols into the WS* protocol stack, is based on such transparent proxies that intercept the SOAP message.

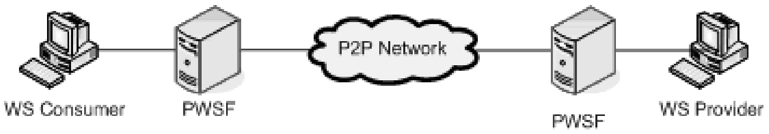

Fig. 6. P2P Web Services Framework (PWSF).

Figure 6 shows a scenario, in which PWSF plays the role of the P2P peer besides the WS provider and the WS consumer. When interacting with the P2P network, a PWSF node appears as a P2P peer and manages to build and maintain the P2P network with other PWSF nodes cooperatively. Therefore, PWSF is more like a gateway joining two networks together. PWSF supports the Plug-in technology, by which a developer builds a software module complying with the plug-in interface and can easily plug the module into the framework to support additional functions. Each PWSF node consists of three layers, the proxy layer, the control layer, and the networking layer. Adjoining layers exchange information via two unidirectional message queues. Each layer consists of two isolated functional components: the framework component and the plug-in. The plug-in contains the specific logical functionality that determines how each layer should behave in a given situation. It is invoked by the framework component when a message arrives or a predefined timer is expired. Then, the plug-in performs consequent actions via the interface provided by the framework component.

5 Evaluation

To evaluate the PWSF performance and scalability a small scale reference system was developed to obtain precise measurements. The reference system consisted of 8 relatively weak PCs (600 Mhz Pentium III, 512MB) that were linked via a dedicated 100Mb Ethernet.

As shown in figure 7 the PWSF nodes perform well until the load increases to ca 20 requests/second. At 30 requests/second they show an added one second latency and two seconds at 50 requests/second. Tests with other PCs show that increased bandwidth and more powerful computing resources allow the delay of the sudden rise in latency e.g. stable response time till ca. 100 requests/second. However letting the ESB distribute the requests (round robin scheduling) over multiple nodes tends to be the most effective way for preventing node overloads.

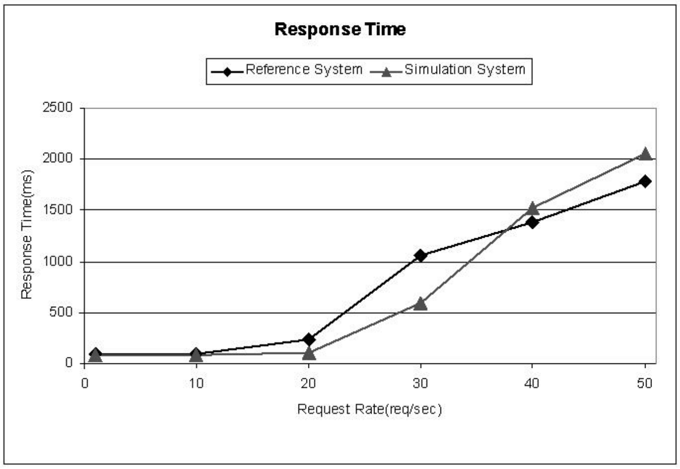

Fig. 7. Response Time.

Using the measurements of the small scale network a simulator was developed. As shown in figure 7 the simulated system tends to behave similar to the reference system. Using the simulator larger networks with 200, 2000 and 10000 PWSF nodes were simulated. The results confirmed the general behavior of Gnutella networks e.g. below 200 nodes (4 connections each) a very good performance and low network overhead can be achieved. (ca. 500 ms). At 1000 nodes 6 seconds delay (average) were observed and above 2000 nodes over 10 seconds latency emerges. Adding and removing (idle) providers at runtime doesn't impact the system in any measurable way as long at the network wasn't split into subnets and the PWSF.

6 Conclusion

Dealing with fluctuating consumer demands requires services to scale up and down depending on the current demand. To achieve such elasticity an organization/enterprise must be able to dynamically add/remove hosts at runtime. While small numbers of hosts can be managed by an ESB, larger numbers require a robust and decentralized approach. P2P protocols like Gnutella are particularly well suited for such a task due to their simple protocol, resilience and good performance. Using transparent proxies as a means to connect WS consumers and providers is an easy and reliable approach. Tests with a reference system and later a simulator showed that the management of up to 200 providers added only a 200ms delay per request.

While the current implementation (PWSF) has shown to be a very robust tool our current focus is on the deployment of other P2P protocols and the use of the programming language ERLANG for the P2P infrastructure.

Acknowledgments. This research has been funded by NSERC and TRlabs.

References

1. Axis v 1.4, http://ws.apache.org/axis/, The Apache Software Foundation.
2. Don Box, "Four Tenets of Service Orientation",[Online]. Available: http://msdn.microsoft.com/msdnmag/isues/ 04/ 01/Indigo/default.aspx
3. Chatarji, J. "Introduction to Service Oriented Architecture (SOA)" [Online]. Available: http://www.devshed.com/c /a/Web- Services/Introduction-to-Service-Oriented-Architecture-SOA, 5 pages. 2004.4.
4. Roy W. Schulte and Yefim V. Natis Service Oriented Architecture, Gartner Reports (SPA-00-7425, SPA-00-7426) 12 April 1996.
5. http://www.w3.org/2002/ws/
6. Business process execution language BPEL v.1.1, Microsoft, BEA, IBM. [Online]. Available: http://www-128.ibm.com/ developerworks/ library/ specification/ws-bpel/
7. Chappell, D.: Enterprise Service Bus. O'Reilly Media, Inc.,Sebastopol (2004)8. Dyachuk, D., Deters, D. "Exposing workflows to load bursts", ICEIS, 2007: 218-225
8. Gnutella protocol specification v4.0 - http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf