Service Architecture Scalability for Sensor Networks

Robin Kirk and Jan Newmarch
School of Network Computing, Monash University, Melbourne, Australia
(robin.kirk, jan.newmarch)@infotech.monash.edu.au

Abstract

The advent of wireless sensor networks and the increasing popularity of the service-oriented computing paradigm mean that ubiquitous environments may finally become a reality. Such environments may contain hundreds to many thousands of services as large scale systems are deployed. The aim of this paper is to test the scalability of two popular service-based technologies, UPnP and Jini, where a large number of services are created.

1. Introduction

Miniature wireless sensors have been developed that can be embedded into any device or inanimate object. Together these sensors can be connected wirelessly to form a hardware base for ubiquitous environments of the future. The computational ability and connectivity of these sensors are rapidly improving as a result of silicone chip manufacturing advances. Although currently sensor networks are mostly non-heterogeneous, in the near future it is possible that existing service-oriented technologies such as Jini and UPnP (Universal Plug and Play) may be used to help manage large numbers of wireless sensors. Utilizing a service-oriented architecture for sensor networks allows platform independent usage and dynamic discovery of sensor functionality.

Where a large number of sensors are connected, performance issues may arise for interested clients, the underlying network and on core features of the middleware itself. To determine the appropriateness of current service-oriented architectures we have performed scalability stress tests on two popular middleware, UPnP from Microsoft and Sun Microsystems Jini.

The structure of this paper is as follows: the next section discusses the two service management middleware, UPnP and Jini. Section three describes and discusses the scalability testing for Jini. Next, the scalability testing for UPnP is described and discussed. Section five compares the results of the testing and concludes with future a discussion of future work.

2. Middleware

2.1 Jini

Jini exploits the mobility of Java code with a service management system tuned towards network realities. It gives service advertisement and discovery, with resilient recovery mechanisms in case of failure. It is interface based, with total flexibility in implementation.
The advantages of this are[1]:

• Jini supplies a service advertisement and lookup registry

• It has inbuilt reflection

• It has an event model

• It supplies a resilient failure mechanism

• It allows flexible proxies, from RPC-like stubs, to "fat" proxies that can use local resources and any appropriate middleware

• It can distribute user interfaces as components of a service

• It can bridge to other middleware systems

• It can handle "legacy" devices through a surrogate model or through Java JNI

2.2 Universal Plug and Play (UPnP)

UPnP enables zero-configuration networking, by allowing devices to automatically join a network and advertise its capabilities as services. Devices may interact with each other on a peer-to-peer basis.

The advantages of UPnP are:

• Device/service advertisement and discovery with no single point of failure

• Standard TCP/IP protocols are used

• Independent of OS/Programming language

There are a number of UPnP implementations written for .NET, C and Java.

3. Jini Scalability

We have tested the system resource requirements for such a large number of objects, to determine the scalability of the framework. A server was written that created a service 10,000 times, created an exporter and proxy and exported the proxy.

In a Jini federation containing thousands of source services, typically a very small percentage would be in use at any one time. The source services not active in a session would lay dormant, waiting to be used. These make individual sources prime candidates for activatable services. Activatable services are only created when a client requests its use, reducing the load and power consumption on the sensor node. Using activatable services requires use of an activation server such as `rmid or phoenix`. While the service is not activated, it still must renew its lease with any lookup service (`reggie`) it has joined. Rather than activating each service to renew its lease, this responsibility can be delegated to a lease renewal service such as `norm`. The remainder of this section focuses on testing activatable source services, and their supporting services.

3.1 Memory Usage

A single Jini service consumes …k of memory.

3.2 Disk Usage

The lookup service `(reggie)` must write any leases to disk, so that it can maintain registered services if it restarts. The lease renewal service `(norm)` must also write to persistent storage any leases it is managing. Figure 5 shows the disk writes for `norm` and `reggie`. The disk writes for activatable `reggie` and `norm` are linear in relation to the number of services; each service writes around 4k and 2.7k respectively. The time taken to register a service with `reggie` and `norm` is the same for the first service and the ten-thousandth service[2]. Although time consuming, this process need only be performed once.

Once services are created, have discovered and joined a lookup service, and registered a lease with a lease renewal service, there is still an ongoing disk usage cost when `norm` renews each lease. Similar leases are batched, causing a large disk usage spike when leases are renewed. The amount of data written to disk is the same amount that was required when registering with `norm`, but it occurs every time a lease expires (about 7 minutes using reggie's default settings). If the time taken to renew the leases exceeds the lease time itself, the hard disk is constantly on. For 1000 activatable services, lease renewal time can take between 2 minutes and 2 seconds, depending on the hard disk access speed. To minimise the frequency of lease renewal the lease time can be extended.

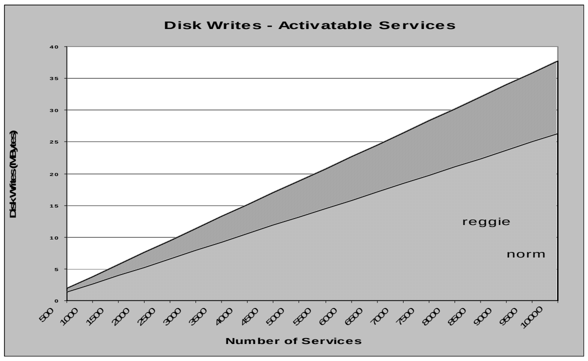

Figure 5. Disk writes for reggie and norm.

3.3 CPU Usage

A problem that does need addressing is lease renewal, as performing this task 10000 times consumes as much CPU as it can, providing the hard disk can keep up. The desktop test system required 12 seconds to renew the leases; this may adversely affect quality of other applications running on this machine.

When a client queries a lookup service for services, and 10000 are returned, it took 25 seconds at 100% CPU for the laptop to add all the service descriptions to a GUI. More CPU efficient methods of renewing leases and displaying available services are needed.

3.4 Bandwidth Usage

The bandwidth usage during service registration was high, for 10,000 services 330,670 TCP packets were captured.

During lease renewal … .packets were captured, this low number is due to the lease renewal server batching the leases.

The controller client produced only 58236 TCP packets, as the service proxies are returned to the client from the lookup service in an array.

4. UPnP Scalability

To simulate an environment containing many UPnP services, a device was created containing 10000 services. Once all the services were created, a GUI control point was created to display the available services.

4.1 Memory Usage

A simple device that contained a power and temperature service running under Windows consumes a maximum of 2.8Mb of memory.

4.2 Disk Usage

Disk usage for the control point was minimal, with only 20,337 Kb being read from disk.

4.3 CPU Usage

The control point took 15 minutes on the laptop to receive the service descriptions from all 10000 services at …. % average CPU usage.

4.4 Bandwidth Usage

For each UPnP service that is advertised two multicast UDP packets were transmitted over the network each of 327 bytes, which occurs on startup and at lease renewal time.

When a control point was started with 10000 services present, over 132,000 packets were captured during the 15 minute period required to discover all services.

5. Discussion

The results show the lookup service of Jini scales far better than the multicast announcement method employed by UPnP. A UPnP client (control point) took 15 minutes to display 10,000 services in a GUI, compared to just 25 seconds for the Jini client.

Disk usage for lease renewal is a bottleneck for a Jini lease renewal service. This can be minimized through the use of a high speed disk. The lease renewal method employed by UPnP produces far more network traffic but does not suffer from slow disk access issues.

The overall results of the scalability tests show that neither Jini nor UPnP scale well when very large numbers of services exist. This is not unexpected, since both middleware were not designed with such large scale systems in mind. Jini is the more scalable of the two middleware providing a high speed disk is used for the lookup service and lease renewal service, and could operate with 10,000 plus services.

6. Conclusion

We have performed scalability tests on two service middleware to evaluate the possibility of their application to a sensor networks.

Keeping large numbers of Jini services active leads to limitations in the number of services possible. Making them activatable allows a much larger number of services. However, some aspects such as lease renewals and UI's show up as potential bottlenecks and may require further work.

New schemes for lease renewal and service discovery are needed to efficiently manage large numbers of sources. Limits in service architecture scalability and techniques to deal with highly dynamic situations need to be explored further.

7. References

[1] J. Newmarch, *A programmer's guide to Jini technology*. Berkeley, Calif.: Apress, 2000.

[2] M. Kahn, C. Della, and T. Cicalese, "CoABS Grid Scalability Experiments," presented at Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS, Montreal, Canada, 2001.

"FUTURE of sensor networks:INTRO"
Qi, H., Kuruganti, P. T. and Xu, Y., "The Development of Localized Algorithms in Wireless Sensor