

# An Autoconfiguring Server-based Service Discovery System

Adam MacBeth

May 11, 2001

## Abstract

*This paper presents the design, implementation, and evaluation of a service discovery system for the one.world pervasive computing operating system. Our system allows clients to find services via a central server and provides an integrated interface for sending events to remote locations. This interface is implemented via a late-binding mechanism, which allows clients to send events directly to the server where they are subsequently sent to the appropriate service. Our system aims to make configuration automatic for the server as well as the client, and this paper presents a leadership election protocol elect a server whenever one is not present. Server performance, election duration, and election scalability are evaluated.*

## 1 Introduction

While ubiquitous computing visions have often been described in recent literature [1, 2], even the most basic of the requisite supporting technologies has yet to be deployed in any usable setting. In the ideal computing environment, devices should be able to interact with one another without static pre-configuration or unnecessary input from the user. In order for this vision to become reality, service discovery (hereafter simply *discovery*), must become an omnipresent technology. While discovery has received a large amount of attention in the research community, and a large number of protocols have been developed in industry, problems of platform and language dependence, insufficient feature sets, and ease of configuration issues have hindered their deployment.

Discovery systems are tasked with providing a way for *clients* to access *services* that they require in an often rapidly changing environment. A service can be defined as any process that provides an interface accessible in the system framework and performs some action deemed useful to some client. The functionality implemented by a service in this

broad definition is therefore unlimited. The real functionality of a service might be implemented as a piece of software, such as an imager transcoder, or a piece of hardware such as a light switch. In either case, the interface to the functionality must be implemented in software to expose it to applications. It is this piece of software that is referred to as a service.

The distinction between discovery systems and directory servers is subtle but important and steadily increasing. Traditional directory services provide simple name to value mappings, are typically manually configured, and in general are not designed to support dynamically changing service sets. Discovery servers, on the other hand, may provide more flexible querying capabilities to describe services, support for dynamic updates, and automatic configuration.

Service discovery systems generally fall into two architectural categories: *server-based* and *server-less*. Server-based systems have one or more servers which accept registrations from services. Clients in these systems query the server to find services. On the other hand, server-less, or peer-to-peer systems are focused on functioning in a more ad-hoc environment where a server may not be present. In these systems, clients simply broadcast their requests, and any service matching a given request responds with more information on how the client can interact with it. One of the most distinguishing characteristics between the two architectures is the need to configure a server. Prior server-based systems have depended on a manually established server, sometimes including manually configured clients as well. While this configuration may be acceptable for enterprise networks, it is certainly not for small ad-hoc networks with rapidly changing membership. Peer-to-peer systems have the advantage here that no configuration is required.

This paper presents the architecture of the discovery system in *one.world*, an operating system for pervasive computing designed for writing highly adaptable applications. Many of the decisions in the design of the discovery system reflect the higher

level goals of the system as a whole, specifically the ability to program for change. I believe discovery to be such a crucial aspect of pervasive computing that it must be integrated with the operating system. This system is the first to integrate discovery as an operating system service.

The target networks for this system are small to medium sized wired and wireless networks where change occurs at the rate of tens of seconds to minutes rather than milliseconds or hours. This range was chosen to match the goals of providing users with pervasive access to information and services no matter where they go. With this in mind, most changes in service availability will occur on a human timescale, because to a large extent many devices are directly associated with humans and therefore undergo significant state changes only when humans manipulate them. For example, when a meeting participant leaves the room, he may carry with him a notebook computer that was providing a service to other participants.

Our architecture is not designed to support millions of devices coming and going on the millisecond timescale. At the other extreme, when the network is mostly static, there is little need for the features that a discovery system offers over a traditional directory. This system may be scalable to larger networks, though this is not an explicit design consideration at this time. The discovery system supports early- and late-binding discovery in anycast and multicast forms and can be configured manually or automatically. The main contributions of this work are the integration of a clean interface to discovery into the *one.world* operating system and the ability of this discovery system to automatically configure itself, allowing developers and administrators to worry less about configuration issues.

This paper is organized as follows. In Section 2, topic of service discovery is discussed, and the features and architectures of a number of previous discovery systems are presented. Section 3 describes the basic architecture of our system, while Section 4 describes the implementation of our service in the context of *one.world*, including the details of autoconfiguration. Section 5 describes the server election algorithm for autoconfiguring the system. Section 5 provides a performance evaluation and sample applications. Section 6 discusses the observed strengths and weaknesses of our architecture, and conclusions are made in Section 7.

## 2 Related Work

This section describes the prior work in the area of service discovery. These systems represent a broad spectrum: from academic to industrial projects, from simple protocols to elaborate server networks, and so on. The most salient features of each system is presented below.

Sun Microsystems' Jini [4] is perhaps the prototypical discovery system, incorporating the roles of client, server, and service. Services register themselves with a server which sends multicast advertisements of its presence to the local network. Clients in turn query the server to find services. The Jini method of querying is based solely on Java types. Clients interact with services through Java events or by downloading Java code which implements the service or allows the client to control the service. Both of these communication methods require that clients and services be implemented in Java or else interact via proxies. Language dependence is strongly integrated into Jini and appears to be a likely reason for lack of deployment.

Berkeley's Secure Service Discovery System (SDS) [10] focuses on building a scalable server infrastructure for providing reliable, secure service discovery. A hierarchy of servers can maintain itself to a large extent, by creating or killing new servers based on demand, but still assumes that a set of secured, dedicated servers is available. The system provides a rich XML-based query and description language which makes this system ostensibly open. However, it also requires the use of Secure Remote Method Invocation (Secure RMI) to perform secure communication, which requires that all participating processes be Java-based.

In the Intentional Naming Service (INS) [12], a self-organizing overlay of servers is established to route service requests through the infrastructure. This model allows for both late and early binding approaches to discovery, and though it does not require the use of multicast, it requires manual configuration of clients to bootstrap them with server addresses.

Universal Plug and Play (UPnP) [8] departs from the systems above in that it is completely serverless. In this sense, UPnP and other purely peer-to-peer systems are autoconfiguring. When a client needs to discover a service, it sends a request message on the local network using IP multicast. Devices which offer services matching the request respond directly to the requesting device using IP unicast. While UPnP is geared toward small home and office networks, it requires support for HTTP, XML,

and IP, which may be restrictive for small devices. Since the protocol is defined using these standards it is programming language independent.

Salutation [13] tries to avoid many of the problems involved in using high-level protocol standards like IP by introducing a Transport Manager which allows for services to be discovered and offered on alternative transport layers such as Bluetooth [3] or IrDA [6], two physical and link layer protocols which both incorporate limited discovery protocols. Salutation can also function with or without a server.

Bluetooth’s service discovery protocol (SDP) can only discover devices which interact on the Bluetooth physical layer. Therefore, a proposal has been made for a mapping from the higher level Salutation API and descriptions to the Bluetooth-specific messages and service descriptions [14].

IETF’s Service Location Protocol (SLP) [9] supports both server-based and peer-to-peer modes of operation, with the goal of functioning in both small and enterprise networks. SLP uses TCP/IP for transport and an open binary data format, which may be more appropriate for resource constrained devices than text formats such as XML.

### 3 Architecture

Since this discovery system was developed in the context of *one.world*, this section provides a brief description of the core aspects of *one.world* (see [5] for more information). *one.world* is an operating system for pervasive computing which aims to provide developers with primitives and services for writing highly adaptable applications. To facilitate simplicity of development, the system exposes 3 core abstractions: *tuples*, which implement data, *components*, which implement functionality, and *environments*, which are containers for tuples, components, and other environments. *Events* are simply tuples which communicate control and data between components in the system and can be sent remotely (across the network), as well as locally. Components export event handlers, which accept events sent from components that import event handlers with a matching type. Linking of imported and exported event handlers can occur at runtime. All resource bindings in the system are maintained by *leases*, which can be canceled, renewed, and inspected by sending appropriate events to the event handler representing a given lease.

An instance of the *one.world* kernel runs in the *root* environment of a given node. Applications run inside environments enclosed in the root environ-

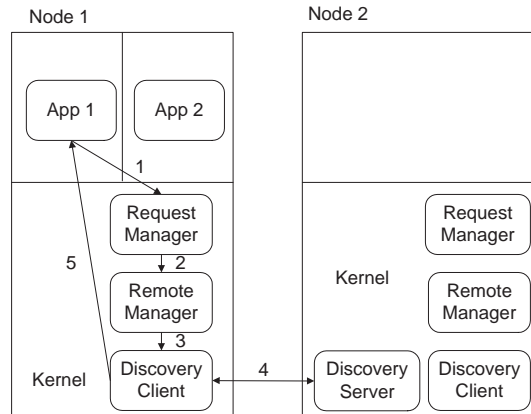


Figure 1: The architecture of the *one.world* discovery subsystem. All nodes contain a client component in the kernel, while server nodes (Node 2) also contain a server component. Numbered arrows indicate the flow of discovery requests between the application and the discovery subsystem.

ment and can only access resources by requesting them from the enclosing root environment. The discovery subsystem runs in the kernel environment because it is privileged to accept messages from all applications running inside the system.

Components of the discovery system include a *client* component and a *server* component. The server component stores service registrations and answers requests for services. The client component is responsible for registering services with the available servers and requesting services on behalf of the application. Therefore, this component implements the inner workings of the client-server protocol so the application does not have to do this directly.

All *one.world* nodes include a discovery client in the kernel. Nodes can be configured to act as discovery servers, or in the absence of such static configuration, a group of client nodes will perform an election to establish a server on one of the nodes. If a node does become a server, a server component is started in the kernel which takes responsibility for the server interactions, while the client component retains all of its basic duties. When the node no longer needs to operate as a server, the server component is unlinked from the kernel. The logic of server election is encapsulated in the discovery client component and thus is always running. A complete picture of this architecture is shown in Figure 1.

In order to enable agile adaptation to change, the system supports *late-binding* discovery, a concept introduced by the Intentional Naming System.

Late-binding means that events can be sent directly to a server where they are then routed to a matching service. In this way resolution and routing of the query are both performed on the server. This is distinguished from the prototypical case where resolution happens on the server and routing is performed by the client on an event-by-event basis, also known as *early-binding*. Because late-binding can route events to any matching service, it acts as an anycast transport. As more than one service may match a given service query, late-binding can optionally support a multicast mode where an event is routed to all services matching the query. It should be noted that late-binding can only be used to communicate with services which are stateless, since each late-bound event may be routed to a different service, depending on the contents of the server at the time of resolution.

### 3.1 Interface

Since *one.world* is focused on providing application developers with useful tools, the interface to discovery was of supreme importance. The interface was designed to be fully integrate late-binding discovery with the Remote Event Passing (REP) mechanism. REP allows *one.world* applications to send remote events to remote nodes by specifying a destination and source of type *RemoteResource* as well as an event to be delivered. The *RequestManager* acts as the intermediary between the application and the kernel, deciding where in the kernel events should be routed. In the case of a *RemoteEvent*, it is first routed to the REP subsystem, encapsulated by the *RemoteManager* component. If the *RemoteManager* cannot resolve the destination and the destination is of type *DiscoveredResource* (a subtype of *RemoteResource*), the event is passed to the *DiscoveryClient* component, which handles interaction with the server.

Early-binding of resources by an application can be performed using *ResolutionEvents*, which are sent to the server for resolution. Upon successful resolution, the server returns a *ResolutionEvent*, including a handle to a service matching the submitted query, or an *UnknownResourceException* if no matching service is available.

The interfaces to services are nothing more than event handlers in this system. *BindingRequests* are sent to the discovery system to request that a given event handler be exported to the discovery server along with a tuple descriptor. As long as the logic of such a service is accessible through a single event handler, it is trivial for the service to be exported

by the discovery subsystem. At the same time, code must be written to handle the interaction between the service and the discovery system. While not yet implemented, I plan to create an abstract service class which takes care of normal operation. Service implementers can then extend this class in the general case or rewrite the service-discovery system interaction for more specialized behaviors.

## 4 Implementation

The *one.world* discovery system is written in the *one.world* framework and as such is implemented in Java. Throughout the implementation, preexisting components of *one.world* made implementation substantially less difficult and less error-prone. For example, the fact that leases exist as a core abstraction in *one.world* allowed the discovery subsystem to be well integrated with the rest of the system. Leases are fundamental to the system because they expose change by allowing the removal of stale information from stores. Upon startup of the *one.world* kernel, a *DiscoveryClient* component is created and linked into the kernel. Given the presence of a configuration option indicating that a node should always act as a server, a *DiscoveryServer* component can also be linked in. The remainder of this section discusses the operation of these two core components as well as the server election process.

### 4.1 Client

The client component is responsible for performing all interactions with the discovery server. A client constantly listens for multicast announcements on a specified IP multicast channel. Upon hearing announcements, the client records the advertised remote event handler for the server and associates it with a lease. The client keeps a list of available discovery servers. When leases expire, the appropriate entry is removed from the server list. When an application attempts to advertise or use a service, the client component chooses a server from its list of discovered servers to interact with. If a given server is unavailable, or the server does not return a matching service, the client will continue to contact servers from the list until the list is exhausted or an appropriate service is discovered. This approach allows clients to interact with a number of non-cooperating servers. For a scalable server configuration, request routing between servers could occur such that clients need only interact with one server. This issue is not addressed further in this paper.

Note that the use of multicast limits discovery of servers to the extent of the local network that is configured to allow multicast traffic. It is our vision that administrators should control the scope of discovery by configuring multicast to operate seamlessly over the area they control. Alternatively, non-administrators can propagate multicast by using application-layer multicast repeaters which pass multicast packets between subnetworks. For ad-hoc network scenarios, such as an impromptu meeting, no multicast configuration is necessary as participants share the same broadcast network.

When exporting a service from an application, the application sends an `ExportRequest` specifying the event handler it wants to export, as well as a tuple describing the service. This tuple descriptor will be stored on the server and thus must include the properties that need to be exposed to other applications. The client component receives the request and must obtain three leases before returning a response to the application. First, the client must obtain a `RemoteReference` for the event handler. In order for an event handler to receive remote events it must be “exported” to the rest of the network. The export process binds the handler to a unique identifier as well as the host and port where the event handler resides. Second, the application must register this exported event handler with the discovery server. Third, the client must obtain a lease which can be given to the application. This last lease allows the application to control how long the client maintains the other lease bindings. The details of this lease are maintained by an instance of the `LeaseManager` helper class. All three of these leases are stored in a table in the client. If any one of these leases expires, the rest of the leases are cancelled as well. As long as the application maintains its lease on the export of its event handler, the client component will maintain the other two leases indefinitely.

The lease established with the server deserves special attention. In order to make remote lease maintenance as easy as local lease maintenance, I modified a preexisting `LeaseMaintainer` helper class to be able to handle remote leases and named this new class `RemoteLeaseMaintainer`. Given a lease event handler and its duration, a lease maintainer will continue to renew the lease until it the lease maintainer is cancel. In the case of the `RemoteLeaseMaintainer`, the event handler is exported as a `RemoteReference`, so lease events must be sent as `RemoteEvents`.

## 4.2 Server

The server component has three duties, handling service registrations and lease renewals, performing lookups, and routing late-bound events. The server advertises itself on a well-known multicast address using periodic `AnnounceEvents` including the `RemoteReference` for the server’s main event handler. Clients can then interact with the server’s event handler using `REP`. Service registration occurs when a client component forwards a `BindingRequest` to the server. Given that there is a one-to-one mapping between `RemoteReferences` and event handlers, duplicate registrations of a `RemoteReference` are not allowed. This prevents two different descriptions of the same event handler should not exist in the server registry at one time. If the `RemoteReference` does not already exist, it is added to the server registry with its associated tuple descriptor and a lease. This lease ensures that the registry does not become full of outdated entries, which could lead to reduced server performance and collapse of the system when applications continually attempt to interact with nonexistent services. The lease handler is exported and its `RemoteReference` and duration are propagated back to the client, which then can respond to the application.

Lookups are performed when the server receives a `LookupEvent` from a client. The server extracts the tuple descriptor from the `LookupEvent`, and proceeds to check entries in the server registry until a matching entry is discovered. Because of *one.world*’s integrated tuples, a highly flexible query language was already available for use by the server, and individual tuples can be checked for matching with the auxiliary `TupleFilter`. Once a matching service is discovered, the `RemoteReference` for the service and the descriptor for the service are returned to the client. If no matching service is found in the registry, a `UnknownResourceException` event is propagated to the application.

The registry on the server is currently implemented as a Java `ArrayList` type, an implementation of a dynamic array. The `ArrayList` type provides no synchronization in any of its methods, which synchronization to be applied only where necessary, but forgoes locking when it does not affect correctness. Instead of ensuring that lookups are atomic, the semantics of lookups are relaxed. Under these relaxed semantics, a service added to the registry after a query begins may not be found by the query. I believe this behavior is reasonable as applications are expected to react to change and should attempt the query repeatedly. As an example of this be-

havior, if another thread modifies the registry while the lookup is occurring by deleting an entry, the lookup may attempt to access an array entry that no longer exists. The solution is to catch the resultant `IndexOutOfBoundsException` and return no entry. While this approach changes the semantics of the lookup, I believe that in a highly dynamic environment, strict semantics are not the best option. There is nothing novel about this registry structure or lookup scheme. Other systems such as INS have used novel lookup schemes, but leave any modifications to the server to future work on improving performance. While performance is an issue of concern, especially for late-binding, it is not our primary focus, though it is discussed Section 5.

Late-binding is implemented in much the same way as early-binding, with the exception that the matching service handle is not returned to the client. Instead, the `RemoteEvent` included in the request is forwarded to the `RemoteReference` in the matching entry, again using REP. When the multicast flag is set in the `DiscoveredResource`, the event is forwarded to all matching services. The source of the forwarded event remains unchanged, so the receiving service can respond to the requesting application if necessary.

### 4.3 Server Election

As mentioned previously, a *one.world* node can be manually configured to act as a server. However, in the general case, it is more appropriate for a network of *one.world* nodes to automatically configure themselves. This autoconfiguration can be effected through an election of one of the nodes to act as a server.

The general method of election is for all nodes to share their *capacity* with the rest of the nodes on the network, with the highest capacity node winning the election. A heuristic function is used by each node to determine this capacity, which is simply a long integer value. This capacity can be thought of as the fitness of a given node to act as the server and can be controlled through a judicious choice of heuristic function. Ideally, the node chosen as the leader should have significant resources to handle the responsibility of acting as a server (including network bandwidth, memory, and processing power) as well as being generally available. While availability can be approximated as the uptime of the node, the two may not be strongly correlated in environments where network partitions occur frequently. Because a poor choice of servers may result in frequent elections and resultant unavailability due to downed

servers, the choice of an appropriate heuristic function is of paramount importance to the efficiency of the system. While the current experimental heuristic functions are based solely on uptime and memory weightings, it seems likely that the heuristic function may need to be modified to obtain the best server choice for a particular network environment.

In general network settings, the distributed leader election problem is quite difficult, and a closely related problem, distributed consensus, has been shown to be impossible in general [15]. The usage scenario I envision for our system allows us to relax a number of constraints which make the former problems difficult. Because this system aims to function over a broad range of networks, the algorithms have not been optimized for a specific case. In fact, for specific settings, our architecture may not be the best choice; this is discussed further in section 6. In the general election problem, there are two main properties of the result which may be considered, uniqueness and selectivity. The uniqueness property states that after an election completes, all nodes will agree on a single node as the leader. Selectivity states that the distributed algorithm should elect the node with the best capacity. While the system should to elect a node that is capable of being a good server, it is not necessary to elect the best server. Of course, the notion of what a good server is may change drastically depending on the network conditions. An ad-hoc wireless network, for example, may experience frequent network partitions such that any node selected as the leader may not be continuously reachable by all other nodes in the network. Therefore selectivity is only of moderate importance. The question of uniqueness is slightly more contentious for us. Because our system is designed to operate over a single multicast capable network, it may be unrealistic to assume that a single server can scale to handling the discovery server responsibilities for the entire multicast domain. For the case of a LAN, however, this assumption is more tenable. Because clients route requests to all available servers, from the point of view of correctness, there is no need to ensure that only a single server exists at all times.

With that in mind, if extra servers are elected and never terminated, the system will become very inefficient with a large number of servers active when only one may be required. In order to eliminate these problems, upon receiving an announcement from another server, a given server will terminate if the other server has a higher capacity. In this way, our system attempts to maintain only a single server while still remaining functional in

the presence of multiple servers. In fact, because servers which are manually established do not terminate upon hearing a better server, it may be a common case for a number of these manually configured servers to exist on the network at the same time. This allows different organizations to establish their own servers, for example.

Next, I describe the original election algorithm that our algorithm is derived from [16, 17]. Nodes first listen for multicast advertisements from a server. If no such advertisement is heard within a given interval (currently a small multiple of the advertisement period), some node will call for an election by multicasting the START message. This message not only indicates that an election should start, but also piggybacks the capacity of the sending node. Nodes that receive the START signal (including the sending node) record the embedded capacity and then proceed to send their capacities to the group. The election terminates after a timeout, which begins when each node receives the START message. Given a reliable and ordered transport method and immediate responses from all nodes, it is clear that all nodes will receive the capacities of all other nodes within a bounded amount of time. If a node finds that it has the largest advertised capacity, it starts the server component. Otherwise, it assumes that another node has been elected as the server and does nothing. In this model, each node assumes that it has the same picture of the network as all other nodes and therefore all nodes come to a consensus on the leader.

Unfortunately, even in the case of a wired LAN, the network is neither reliable nor ordered, and processes do not respond immediately, as they may experience processing delays, intermittent failures, and network contention. In this scenario, using the simple algorithm described above results in possibly inconsistent views of the network by different nodes. Our algorithm therefore assumes that these different views can occur. Because servers advertise their presence, the nodes need not agree on a single leader. Instead, each node simply decides whether it is a server or not. The advertisements contain a *RemoteReference* to the server, so a client can discover the identity of the server shortly after an election completes. This algorithm may result in more than one server being established, but it will never result in a situation in which no leader is elected. Since each node hears its own advertisements, it will elect itself in the absence of any other advertisements.

A simple summary of our algorithm for each node is as follows:

1. Listen for server announcements. If none are

heard within a specified period, call for an election by sending a multicast START message including the local capacity.

2. Upon receipt of a START message, start a timer to bound the election. Record the capacity of the message. Send a multicast message with the local capacity.
3. Upon receipt of a CAPACITY or START message, record the capacity if it is higher than the highest capacity seen so far. Repeat Step 3 until a timeout occurs.
4. When the timeout occurs, if the highest recorded capacity is equal to the local capacity, start the server component.

If two nodes have the same capacity, the node with the highest IP address wins (packets are sent using IP multicast). While it may be possible to use a more complicated algorithm to ensure uniqueness, the greater concern is bounded start-up time. Therefore I chose fixed time for the duration of the election. The selection of this time bound is discussed in the Evaluation section.

Multiple nodes may call for an election at the same time, but since capacities are embedded in START messages, there is no need to ensure that there is a unique calling process. The first START message is used to start the timeout and to receive a capacity, while any subsequent START messages are treated simply as capacities announcements.

This algorithm is open to spoofing. For example, denial of service attacks can be performed by injecting messages advertising a capacity larger than that of any other nodes. If the injector of these messages has no intent of acting as a server, it may be elected as the only server and can subsequently prevent service discovery from progressing. However, in the local environment which I envision for deployment, cooperation among nodes is assumed.

## 5 Evaluation

This section presents an evaluation of the the performance of the discovery system as well as an analysis of the scalability and responsiveness of the election process.

### 5.1 Server Performance

In order to investigate the performance of this server, I measured the latency and throughput of

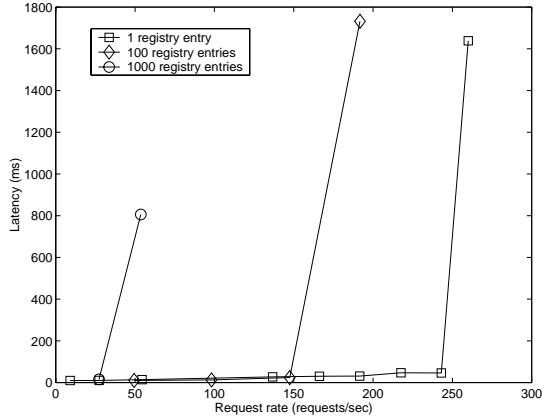


Figure 2: Plot of latency versus request rate for various registry sizes. Smaller registry sizes achieve higher throughput before seeing latencies increase dramatically.

a server when presented with varying rate early-binding request workloads. The server registry was first populated with dummy entries to create a registry of the appropriate size. A target entry was then added to the end of the registry to ensure measurement of the worst-case times for a given registry size. Three client machines were used to generate a load on the server. The requests contained a simple string equality query for the target entry. Client machines generated requests periodically in bursts, with the slowest speed being 1 request every 100 ms. To measure the latency, packets were periodically timestamped. When resolution of a given request was seen by a client, if it was timestamped, the difference between the timestamp and the current time was reported as a latency measurement. Reported numbers are averages of 50 measurements (which are themselves time averages over periods of approximately 1 second). Tests were performed with client machines containing 500 MHz and 800 MHz Pentium III processors, running Linux 2.2.14 on 100 Mb/s switched Ethernet. The server was an 800 MHz Pentium of a similar configuration.

The results for the workload are presented in Figure 2 with the standard deviation results shown in Figure 3. The number of requests per second was calculated as the aggregate of the requests rates from the individual clients.

As request rates increase, latencies increase as well due to queuing on the server. At higher request rates, the latency increases dramatically, to the point where the server becomes unusable. The same behavior is observed for all registry sizes. Standard deviations increase along with latencies, showing

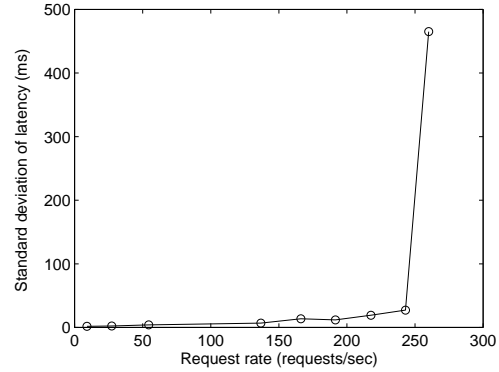


Figure 3: Plot of standard deviation of latency for the 1 entry registry data of Figure 2. The standard deviation increases nearly identically to the latency data.

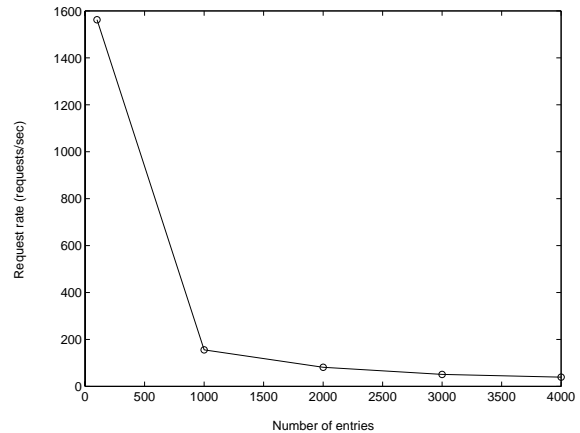


Figure 4: Measurements of the request rate supported by registries of different sizes.

that at high request rates, response times are increasingly variable. The maximum attained request rates are 243, 147, and 27 requests per second for registry sizes of 1, 100, 1000, respectively. In order to investigate the breakdown of these latencies, the latency of lookup operations were made directly on the server. The time to perform 1000 lookups was measured as a function of increasing registry size, from 1 to 5000 entries (Figure 4). While the server latency measurements do not exceed these numbers, they do not come close to achieving the raw request rate of the registry.

Separate measurements for late-binding were not taken. Query performance should be identical to these measurements, though some extra latency may be incurred by large packets. An interesting measurement which could be made is the through-

put of the server as a function of RemoteEvent payload size (the size of the enclosed Event) and registry size.

## 5.2 Election Duration

As mentioned in the description of the election algorithm, the bound on the length of the election must be chosen to make the election complete efficiently. This choice involves a tradeoff between speed (responsiveness) and completeness (avoidance of multiple competing servers). If a server is not available and a client needs to acquire a service, it can not do so until a server has been elected and populated with services. Therefore, the election time should be chosen to be as short as possible.

On the other hand, the bound needs to be chosen so that the election has a high probability of reaching full completion within the time chosen. “Full completion” means that all nodes have received the capacities of all other nodes and can make a fully selective and unique choice of leader. Because of the attrition process described previously, it is not essential that the election run to full completion. However, the danger in not running to full completion is that the presence of a large number of servers may cause chaos on the network, leading to further delays. Clients will attempt to register with a large number of servers, therefore generating a large number of unnecessary leases on the client side. Advertisements and registrations of multiple servers will consume a larger fraction of the network bandwidth. The time bound could be made very long to achieve full completion most of the time, despite an unknown number of nodes on the network. This choice would greatly harm responsiveness, however.

In general, the majority of nodes on the network will respond within a reasonable amount of time, leading to a situation in which most nodes on the network have a similar picture of the network. For example, if one node has advertised a much higher capacity than any others, and most nodes have heard this node, then most of these nodes will defer to the higher ranked node.

I now derive the bound which is used in this system using these assumptions. We need to calculate the total length of time it takes for a calling process to send an announcement and all other processes to respond. Since we assume a broadcast medium, only one process can send at a time. To express transmission delay for an single response, we will use the equation:

$$latency = propagation + transmit + queue \quad (1)$$

where *propagation* is the speed of propagation over the physical medium, *transmit* is the time to transmit the given amount of data over the network, and *queue* is the queuing delay in the network [18]. In addition to this latency of transmission, we also consider latency in response, which includes network contention and processing delay. We assume that the *queue* term above is zero given that we are using a broadcast network. In switched networks, there may be some switching delay, but because this may vary greatly depending on the network configuration, we ignore it for now. Likewise, we assume that processes do not delay before sending.

The total latency for the election can be expressed as:

$$total\ latency = (N \times latency) + contention \quad (2)$$

where  $N$  is the number of hosts on the network, *latency* is defined as in equation 1, and *contention* is the amount of contention for all hosts. The ideal time bound is equal to this number, but when this number is too long to obtain acceptable responsiveness, a shorter time bound may be required.

I now present an example of the election duration on an Ethernet. The delay due to contention is dependent on the number of hosts on the network. In this case we assume that the network has 256 hosts (since most Ethernets coincide with IP class C subnetworks). Previous studies of Ethernet have attempted to quantify the transmission delay experienced for different numbers of sending hosts and packet sizes. Since I do not have the resources to measure the actual contention times for this system, I use the data from a study of Ethernet performance [19]. The data of interest from this study are those showing the average delay in transmission when a given number of hosts are sending at the same time. The study only includes measurement up to 23 hosts, but as the data are linear, interpolation is trivial. The data show a linear plots of average transmission delay versus number of hosts. For a 768 byte packet (the closest to the election packet size) versus number of hosts, the estimated equation is

$$delay = \frac{18}{23}n$$

Computing *delay* for  $n = 256$ , the number of hosts, yields a average delay of 200 ms. Similar interpolation of the plot of standard deviation versus number of hosts gives

$$delay = \frac{58}{23}n + 20$$

predicting a value of 665 ms for 256 hosts. Assuming that 99% of all times lie inside 3 standard deviations, we can predict that all hosts will send their message by time  $200 + (3 \times 665) = 2195$  ms.

Assuming 10 Mb/s Ethernet and 700 byte election messages, a single message takes a maximum of  $51.2 \mu\text{s}$  to propagate [18] and  $70 \mu\text{s}$  to transmit, for a total transmit latency of  $121.2 \mu\text{s}$ . The transmission delay for all hosts is therefore  $256 \times 121.2 \mu\text{s} = 31$  ms. Using equation 2, the total delay is therefore  $31\text{ms} + 2195 \text{ ms} = 2226$  ms, or a little over 2 seconds.

It should be noted that this analysis uses very conservative numbers; the average case may be much better than this. Smaller numbers of hosts and more realistic network configurations will also bring down the time required for this election. Nonetheless, this result is quite reasonable; if we choose the time bound to be equal to this total latency, there should be no problem supporting human-timescale operation.

Another factor in achieving human-timescale operation is the time it takes to call for an election. Servers in the system broadcast periodically, and if clients do not hear this announcement after 2 time periods, an election is called for. The current announcement period used is 1 second, and therefore, the detection time is 2 seconds. The announcement period could be increased, but this would increase both the amount of bandwidth utilized as well as the number of cycles utilized on the server for this activity. As shown in Figure 5, a period of 1 second places the bandwidth utilization right at the knee of the curve. This value is 5712 bps, a small fraction of the bandwidth of a 10 Mb/s Ethernet. While this value is quite insignificant, decreasing the period much lower would result in a drastic increasing in bandwidth. This system has not been used enough in dynamic situations to know if these time values lead to good responsiveness for general human activities, but the bandwidth utilization appears reasonable using this period.

### 5.3 Election Scalability

An issue related to the selected time bound is how the election algorithm scales to larger and smaller networks. As a brief example, a network with a lower bandwidth than Ethernet, such as Bluetooth at 1 Mb/s, might not be able to support as many devices as Ethernet due to its lower bandwidth. This lower bandwidth would mean longer transmit times and therefore longer total delays. Fortunately, the number of devices on a single Blue-

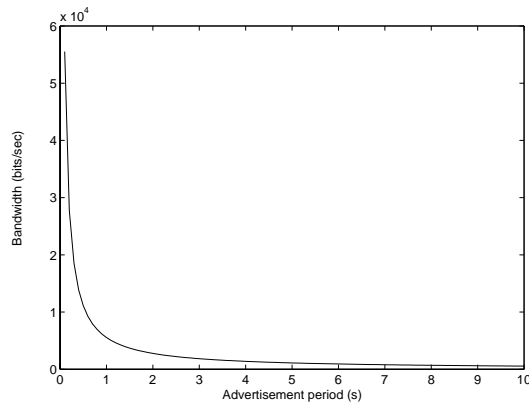


Figure 5: Bandwidth utilization for server announcements with a 714 byte packet size.

tooth network is much smaller than that on an Ethernet, so the required time for full completion of the election is proportionally shorter.

If the target network is configured so that IP multicasts can pass between attached subnetworks, the problem is no longer simply that of a single broadcast network. In order to show how this algorithm scales with latency and number of hosts, consider equation 2. While contention time is likely to be less of a factor, the overall latency may be larger due to further physical separation of the hosts. The real tradeoff in scalability is therefore between the delay and the number of hosts. In order to keep the election time constant, if one parameter is increased, the other must be decreased. Let us assume that the time bound is held constant at 1 second. With 1000 hosts, we can afford a delay of only 1 ms per sending host if we want to stay within the time bound. On the other hand, a truly wide area network, which might have multicast delays of 100 ms, could only support 10 hosts within this time bound. Of course, with wide area multicast, there may be further issues of logistics. This analysis shows that the system is best suited to those conditions for which it was designed: small to medium ad-hoc and enterprise networks.

If we are willing to accept less than full completion of the election, we can keep the election time constant but increase the number of hosts or latency. In the most degenerate case, the election time is negligible compared to the time required for a full election and every participating node becomes a server for a short time before hearing other servers. Both of these properties apply to wide-area networks, a domain in which our system is not intended to be used and obviously will perform poorly.

## 6 Discussion

This section presents some of the problems with this service discovery system, unresolved issues, and areas for future work.

In many ways, server-less architectures are better able to support rapidly changing networks where machines are constantly coming and going. In the server-based approach, if a server goes down, an election must be performed and a new server elected. This is costly both in the election time and the time to repopulate the server. If the elected server goes down soon after election, further delays are seen by applications. While the choice of a good heuristic function appears essential to maintaining server availability, it is as yet unclear what properties reflect a good candidate or whether these properties are easily measurable. Servers act as a single point of failure no matter how good the choice of server. Therefore, the effectiveness of a server-based approach hinges on the usefulness of the properties it provides that a peer-to-peer system cannot. I believe that these properties include a flexible query language and late-binding.

Peer-to-peer systems, notably UPnP, provide very simple query functionality with the assumption that small, resource constrained devices will be overburdened by incorporating a more complex query language. While this assumption may be unreasonable, I believe it is a useful point of comparison as an existing system. The ability to perform flexible queries is likely to form an essential role in service discovery. Jini provides the ability to query based only on the Java type of the service requested, severely limiting the ability of applications to describe the properties of the required service without *a priori* knowledge of a specific service type.

While I have shown that the times involved in the election are reasonable, I have not studied any applications in daily use. In highly ad-hoc interactions, establishment of the server, population of the server with services, and subsequent discovery of services may take too long to provide users with the responsiveness they need. With the current protocol, two isolated devices (say handheld computers), will each elect themselves as a server and then populate that server with services from that node. When the two devices are brought into close enough proximity for ad-hoc networking between them, the server with the lower capacity will terminate, leaving one device running a server fully populated with its services, while the other device must quickly register its services with the other.

A modification to the protocol could allow de-

vices to retain their server component and local service registrations, while not advertising the server to the entire network unless elected under the normal election process. While this is useful for applications to find services running in other environments on the same node, it could also unnecessarily burden small devices by forcing them to run both the client and server simultaneously at all times. Another alternative would be to allow devices to operate in a peer-to-peer fashion when necessary and in a server-based mode when possible. However, it is unclear how late-binding would be implemented in an efficient manner in the peer-to-peer side of this system.

The usefulness of late-binding is still under debate. While I believe late-binding is essential for helping applications deal with change, I do not have a large base of applications which have used this technology to their advantage. While late-binding is not a performance enhancement in the general case, it may speed performance when service availability is changing rapidly. For example, an application which queries a service once a minute for data may experience delays if the service is moving at periods less than one minute. In this case, early-binding would require the application to attempt to contact the service, receive the failure after a timeout, perform a new lookup at the server, and finally contact the service directly. All of this could be contained in just one late-binding request. Late-binding therefore removes much of the burden of failure handling from the application and places it on the server. As previously noted, in its current incarnation, late-binding can only be used for services which can be considered stateless. While this works fine for the service described, more complex applications may require services to maintain state between interactions. We do have evidence that programmers find late binding useful. A number of students who have used *one.world* for an experimental course project found that late binding was very useful in writing their application. I believe this was partly due to the fact that the programmers were forced to write less code to use late-binding than early-binding.

There are currently few applications that actively use *one.world*'s service discovery features, so it is difficult to quantify the required performance. Nonetheless, given similar measurements from the INS system, which showed between 700 and 900 requests per second in a Java implementation, our system falls behind. This is one of the first performance measurements made of a system written for *one.world*, so these measurements may also be indicative of the performance of the underlying *one.world*

core. The first place to start looking to improve performance will be the registry data structure and lookup procedure which currently performs a linear-time search.

While the server is implemented to return the first match in response to queries, it may also be useful to provide a facility for best matching. The semantics of this approach are slightly stronger than anycast, as requests now specify a specific field to . The definition of what is the best can be defined on a type-by-type basis, but may be unclear for a generic type. The Intentional Naming System provides this functionality, but it is unclear how important this is to applications. Given that performance appears limited at this time, it seems unwise to add additional features which might heavily load the server.

This system needs to be tested on a variety of networks to investigate whether the protocols transfer well, from a wired to a wireless network for example. As this architecture has not been optimized for a particular network type, it will be instructive to see where performance or feasibility problems occur. The system will likely need to be deployed and used on these networks with a number of different applications in order to see its weaknesses. Past work has looked at optimizing service discovery in Bluetooth networks to account for the properties of the underlying physical layer [11], so this work might make a good comparison.

Currently, all communication with discovery servers is through *one.world* Events. While this limits the system to interactions among *one.world* capable devices for the time-being, when XML object serialization is added to *one.world*, this limitation will disappear. While this will still require devices using the system to be XML-capable, this greatly expands the platforms that will be able to use the *one.world* discovery system. The system is also currently dependent on TCP/IP for networking. Given our choice of implementation in Java, access to a variety of networking technologies is limited.

A final issue is that of service description ontology. There needs to be a common way for application and service writers to rely on each other to use common terms for description. For example, if an application needs to find a device for displaying a document, the application could request a “display”, “monitor”, or “screen”, but if the service does not use one of these terms in its description, the application will never find it. While I do not attempt to answer such questions here, this is certainly an area which will require future pursuit if pervasive applications are to work well.

## 7 Conclusion

This paper has presented an architecture for a server-based discovery service for local area networks. Our system incorporates features from previous discovery systems which I believe are essential to an effective service discovery system. I chose a server-based architecture in order to provide support for a flexible query language and late-binding multicast and anycast. This choice appears to be in opposition to the goal of ease of configuration and adaptability, given that a server must be established and services registered with it for discovery operations to occur. Therefore, I presented a method for a number of *one.world* nodes on a local network to automatically elect a server. This system of autoconfiguration was designed to scale to a moderate number of nodes on a local network, and I have shown its feasibility in this arena. More work is necessary to determine the effectiveness of this approach under various types of network environments.

## 8 Acknowledgements

I am indebted to the core members of the *one.world* team: Robert Grimm, Janet Davis, Eric Lemar, and Steve Swanson. Special thanks goes to my faculty advisors on this project, Gaetano Borriello and Steve Gribble.

## References

- [1] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, September 1991.
- [2] M. Esler, J. Hightower, T. Anderson, and G. Borriello. Next century Challenges: Data-centric Networking for Invisible Computing. The Portolano Project at the University of Washington. In *Proc. of the Fifth ACM/IEEE Intl. Conf. on Mobile Networking and Computing*, August 1999.
- [3] Bluetooth SIG, *Bluetooth Specification, Version 1.1*, <http://www.bluetooth.com>, 2001.
- [4] Waldo, J. Jini Architecture Overview. *Sun Microsystems Whitepaper*, 1998.
- [5] *one.world* website. <http://one.cs.washington.edu>.

- [6] Infrared Data Association. Technical Summary of "IrDA DATA" and "IrDA CONTROL", <http://www.irda.org>, April 20, 2001.
- [7] Home Audio Video Interoperability White Paper, <http://www.havi.org>, 1999.
- [8] Universal Plug and Play Forum. Universal Plug and Play Device Architecture, Version 1.0. <http://www.upnp.org>, June 13, 2000.
- [9] E. Guttman, C.E. Perkins, and M. Day. Service Location Protocol, Version 2. Internet RFC 2608, June 1999.
- [10] S. Czerwinski, B. Zhao, T. Hoddes, A. Joseph, R. Katz. An Architecture for a Secure Service Discovery Service. In *Proceedings of MobiCom '99*, Seattle, WA, August 1999. ACM.
- [11] B. Raman, P. Bhagwat, and S. Seshan, Arguments for Cross-Layer Optimizations in Bluetooth Scatternets, *Symposium on Applications and the Internet (SAINT'01)*, Jan 2001
- [12] W. Adje-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The Design and Implementation of an Intentional Naming System. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 186-201, Charleston, SC, 1999.
- [13] Salutation Consortium. White Paper: Salutation Architecture: Overview. <http://www.salutation.org/whitepaper/originalwp.pdf>, 1998.
- [14] R. Pascoe and B. Miller. Mapping Salutation Architecture APIs to Bluetooth Service Discovery Layer, Version 1.0. <http://www.bluetooth.org>, July 1, 1999.
- [15] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
- [16] J. Brunekreef, J.-P. Katoen, R. Koymans, and S. Mauw. Design and analysis of dynamic leader election protocols in broadcast networks, *Distributed Computing*, vol. 9, no. 4, pp. 157–171, 1996.
- [17] C. King, T.B. Gendreau, and M.N. Lionel. Reliable Election in Broadcast Networks, *J. Parallel and Distributed Computing*, vol. 7, pp. 521-540, 1989.
- [18] L. L. Peterson and B. S. Davies. *Computer Networks: A Systems Approach, Second Edition*. Morgan Kaufmann, 2000.
- [19] D. Boggs, J. Mogul, and C. Kent, Measured Capacity of an Ethernet: Myths and Reality, *WRL Research Report 88/4*, Western Research Laboratory, September 1988.