Neuron — A Wide-Area Service Discovery Infrastructure

Hung-Chang Hsiao     Chung-Ta King*
Department of Computer Science
National Tsing Hua University
Hsinchu Taiwan, 300
hsiao@pads1.cs.nthu.edu.tw

Abstract

A wide-area service discovery infrastructure provides a repository in which services over a wide area can register themselves and clients everywhere can inquire about them. In this paper, we discuss how to build such an infrastructure based on the peer-to-peer model. The proposed system, called Neuron, can be executed on top of a set of federated nodes across the global network and aggregate their resources to provide the discovery service. Neuron is self-organizing, self-tuning, and capable of tolerating failures of nodes and communication links. In addition, it allows the services to be described with arbitrary forms and the system load to be distributed evenly to the nodes. Neuron also supports event notification. We evaluated Neuron via simulation. The preliminary results show that service registration, discovery and service state advertising take at most $O(\log N)$ hops to complete.

1. Introduction

Due to the ubiquity of the Internet, the maturity of wireless networking and the availability of portable and embeddable thin devices, billions of devices, ranging from multi-tera FLOPS supercomputers to miniature sensors can be interconnected and function together to provide a wide variety of services. Services, no matter where and how they are provided, may be accessed from any place, at any time, via any device. However, with the vast amount of information and resources in the Internet, there is no easy way for a user or program to know all the available services, not to mention to track them.

The problem is further complicated by the fact that the Internet is dynamic, spontaneous, unstable, and unreliable. The deployed services may become unavailable, because the devices or communication links that provide the service may fail. New services or capabilities may be added at any time, and their states may change dynamically. Worse, users may roam about freely and enter environments that are totally strange to them. Discovering services in the new environment becomes a problem. All these point to a need to name and discover services in a robust way, particularly at a global scale.

The task is challenging not only because of the amount of information needed to be tracked, but also because of the dynamic nature of the Internet. Centralized discovery serv-
cover a service, only $O(\log N)$ hops of communication are required, where $N$ is the number of active nodes in the system and a hop is one end-to-end communication between any two nodes. (6) Neuron can notify the changes of states of services that the end systems are interested in. (7) The load in Neuron can be evenly distributed to the participating nodes while minimizing the utilized resources. (8) Neuron is an application-level solution and thus avoids the need for cumbersome deployments and OS-level supports. It does not need centralized administrative control. To our knowledge, Neuron is the first and the most comprehensive work towards P2P-based service discovery infrastructure.

The remainder of the paper is organized as follows. Section 2 presents the system goals of Neuron. The design of Neuron is detailed in Section 3. Section 4 presents our evaluation methodology and reports the evaluation results. Related works are presented in Section 5 and conclusions of the paper are given in Section 6, with possible future research directions.

2. System Goals
With the ubiquitous Internet, it is possible to aggregate the untrusted or federated resources (e.g., desktop PCs, workstations, and servers) on the Internet to provide distributed and global-scale data and computation services. Neuron is an overlay infrastructure on top of Tornado for service discovery services. Each participating node contributes portion of its storage and computation to Neuron to help discover resources in the system. A participating node can be a client to request particular services from other nodes in Neuron, and at the same time be a server to supply the stored services to other nodes.

Neuron was developed with the following goals in mind:

- **Aggregating resources on the network**: The Internet today has interconnected an enormous amount of computing and storage devices. Resources on these devices, e.g., CPU cycles and storage, are often underutilized. Aggregating these devices not only exploits their idle resources but also allows rapid deployment of desired services.

- **Reliability**: Resources exploited on the Internet may be highly volatile, depending on the environment deployed. For instance, the aggregated resources may be attacked (e.g., denial-of-services attacks) and become unavailable, or they may be linked to the Internet through intermittent wireless links. Our infrastructure must accommodate such unstable resources and tolerate possible faults.

- **Self-organization**: Neuron should admit nodes from different administrative domains and allow them to dynamically participate and leave the system. Each node should manage its connectivity to the system and help other nodes to perform their requests without centralized control.

- **Load distribution**: To prevent performance bottlenecks in the infrastructure, Neuron has to distribute the system load to participating nodes according to their capabilities, e.g., computation, communication and energy. When service objects are first generated, they should be allocated to the nodes evenly. Popular service objects are epidemically cached and replicated by multiple nodes to alleviate the load imbalance.

- **Security**: Nodes coming from different administrative domains may not be trusted. A malicious node may peek at data items in the infrastructure and alter them. Neuron needs to provide encryption to data to reduce the risk of malicious attacks. Only those nodes with legal keys can access published service objects.

- **Durability**: Since nodes may dynamically leave the system, the published services maintained by those nodes might not be available. To guarantee that every service object is accessible at any time, Neuron should provide high service availability.

- **Scalability**: The entire state of the system should be distributed to the nodes to achieve scalability. The amount of partial states maintained by each node should not scale to the system size.

- **Responsiveness**: Neuron should provide its discovery service promptly. The number of requests and the system size should not influence the response time significantly. In addition, services may change their states to reflect their current situation; these changes should be presented as agile as possible.

- **Characterization**: The service can be described in free form, and clients can discover their desired services via free-style description. Meanwhile, any matched service needs to be responded to clients in fidelity.

3. Neuron
Neuron is built on top of Tornado [5] and supports efficient service registration and discovery. In this section, Neuron is introduced. We first give an overview of Neuron in Section 3.1. Section 3.2 presents basic operations in Neuron. Operations that allow Neuron to adapt to the dynamic environments are described in Section 3.3. Section 3.4 discusses the reliability issues in Neuron and the scalability issue is addressed in Section 3.5.

3.1. Overview
Conceptually, Neuron constructs a *virtual shared space*. Service providers can register their services through the virtual shared space by using free form expressions. Clients can look up the registered services and query service states via similar expressions. Service descriptions are encrypted via the RSA algorithm, for example. Clients intending to acquire a service need to obtain the corresponding public keys first, and only service providers can alter their services with private keys.

Neuron is implemented on top of Tornado and thus built with the hash-based addressing scheme. Neuron is therefore capable of supporting nomadic services by decoupling service location and address. The hash-based scheme also allows the virtual space created by Neuron to support anonymity, i.e., service providers and clients are not necessarily aware where to register and access services. It leverages the overall system reliability and service durability by reducing malicious attacks on particular nodes in Neuron. Applications can thus concentrate on what services they want rather than where and how to accesses services. The states of the services can be promptly updated to the clients that are “interested” in such services.

Figure 1 shows a usage example of Neuron. Note that with the aid of Tornado, Neuron is able to operate in the
global network. The example here, which shows service discovery in a localized setting, illustrates the diversity of Neuron. In this example, there are three services presented in the virtual shared space: telephone, printer, and coffee vending machine. Bob, Mary and Paul would like to have a coffee. Therefore, they request a nearby coffee vending machine from the virtual space by issuing the lookup operation. After the coffee vending machine is discovered, Bob, Mary and Paul may like to know what coffee drinks are available. Thus, Bob, Mary and Paul subscribe to the services provided by the discovered vending machine (see Figure 1(b)). The current state of the vending machine will be published to them (see Figure 1(c)). Suppose further that Paul likes to have a Cappuccino, but the machine runs out of it. The subscription allows Paul to be notified once the drink is refilled. Note that if Bob, Mary and Paul would like to cancel their subscriptions, they need to explicitly unsubscribe them.

3.1.1. Naming

Naming in Neuron adopts a free form expression. Each service is characterized by \((service\ name, attribute_1 = value_1, attribute_2 = value_2, ... , attribute_n = value_n)\), where \(service\ name\) is the type of the service and \(attribute_i = value_i\) indicates the associated attributes. For example, a color display in room 734 of the EECS building can be characterized as \((LCD\ Display, color\ depth = 65536, resolution = 1024\times 768, room = 734, building = EECS)\).

To register a service to Neuron, a request is sent to its service home node. The service home node is a Tornado node and is responsible for providing a permanent storage for the service and the associated states. It also provides access control for the service. Given a service, its ID is calculated via the following hashing function:

\[
H(H(service\ name))
\]

where \(H\) is the hashing function used by Tornado. Given the ID of the service, it can then be mapped to a Tornado node, which is called the service home node of the service.

3.1.2. Leasing

Neuron requires each registered service provide a time contract, which denotes the service’s lifetime. The lifetime is determined by the service provider. If a registered service does not renew its contract, this implies to Neuron that the service may be failed or terminated. Its entries will be removed from the virtual shared space. An “active” service is one that periodically or aperiodically advertises its states to Neuron. Neuron guarantees a permanent and durable home node to host the active services. The leasing model allows a service to present its most up-to-date states to the clients.

3.2. Basic Operations

In this section, the data structure used by Neuron is first presented in Section 3.2.1. Section 3.2.2 then describes the operations for service registration, discovery, subscription and renewal.

3.2.1. Data Structure

Assume that a service is expressed as \((e_1, e_2, e_3, ..., e_n)\), where \(e_i\) is the hashed value of the service name (i.e., \(e_i = H(service\ name)\)), and \(e_i = H(attribute_i = value_i)\), where \(2 \leq i \leq n\). For simplicity of presentation, we assume that \(e_2 < e_3 < ... < e_n\). The ID of the home node is then given by Equations 1 as \(H(e_i)\). Figure 2 shows an example in which two services, the printer and the vending machine, are registered to the virtual shared space. The printer is expressed as \((X_1, X_2, X_3, ..., X_n)\) and the vending machine as \((Y_1, Y_2, Y_3, ..., Y_m)\).

The data structure used by Neuron is a Neuron tree. A Neuron tree is a virtual binary tree, and each service is associated with one such binary tree. The root vertex of a Neuron tree is stored in the home node of the service. The home node not only provides a permanent store and access control for a service object, but also computes an initial binary tree for it.

The home node calculates the IDs of the leaf vertices of a Neuron tree by applying the hashing function.
The Neuron tree is denoted as 

\[ H(X_1X_2...X_n) \]

where \( k_i = 0 \) or 1, \( i = 2, 3, ..., n \) and \( \oplus \) denotes the concatenation. Thus, the IDs of the leaf vertices are \( H(e_2), H(e_3), H(e_4), ..., H(e_n) \). As Figure 2 shows, the leaf vertices of the printer service are those nodes with IDs \( H(X_2), H(X_3), ..., H(X_n) \). Given the ID of a leaf vertex, it can be mapped to a Tornado node, where it is stored. That Tornado node will be referred to as a leaf node. At a first glance, it seems that the tree requires at least \( 2^{n-1} \) Tornado nodes to store. However, these Tornado nodes need not be disjoint, and it actually depends on the mapping from the data addressing space to the node addressing space in Tornado and the load of the Neuron tree (i.e., if the load is continuously increased, more Tornado nodes can be introduced to the Neuron tree to alleviate the load of participating Tornado nodes). In addition, the number of terms in an expression, \( n \), is not expected to be large. Further discussion of this point can be found in [4].

It is possible that several Neuron trees share the same leaf vertices. That is, they have common attribute-value pairs in their service expression. To distinguish the Neuron trees that have common attribute-value pairs, each Neuron tree is associated with a unique tag, which is the hashing key (i.e., \( H(e_1 \oplus e_2 \oplus e_3 \oplus ... \oplus e_n) \)) of the service object.

In addition to calculating the IDs of the leaf vertices, the root node also computes the IDs of the intermediate vertices in the Neuron tree, which are called the forwarding vertices. The forwarding vertex at the \( i \)-th level and \( m \)-th position in a Neuron tree is denoted \( H_{im} \), and its ID is obtained as follows.

\[
E = \{k_2e_2 \oplus k_3e_3 \oplus k_4e_4 \oplus ... \oplus k_ne_n\},
\]

where \( k_i = 0 \) or 1, \( i = 2, 3, ..., n \) and \( \oplus \) denotes the concatenation. Thus, the IDs of the leaf vertices are \( H(e_2), H(e_3), H(e_4), ..., H(e_n) \). As Figure 2 shows, the leaf vertices of the printer service are those nodes with IDs \( H(X_2), H(X_3), ..., H(X_n) \). Given the ID of a leaf vertex, it can be mapped to a Tornado node, where it is stored. That Tornado node will be referred to as a leaf node. At a first glance, it seems that the tree requires at least \( 2^{n-1} \) Tornado nodes to store. However, these Tornado nodes need not be disjoint, and it actually depends on the mapping from the data addressing space to the node addressing space in Tornado and the load of the Neuron tree (i.e., if the load is continuously increased, more Tornado nodes can be introduced to the Neuron tree to alleviate the load of participating Tornado nodes). In addition, the number of terms in an expression, \( n \), is not expected to be large. Further discussion of this point can be found in [4].

It is possible that several Neuron trees share the same leaf vertices. That is, they have common attribute-value pairs in their service expression. To distinguish the Neuron trees that have common attribute-value pairs, each Neuron tree is associated with a unique tag, which is the hashing key (i.e., \( H(e_1 \oplus e_2 \oplus e_3 \oplus ... \oplus e_n) \)) of the service object.

In addition to calculating the IDs of the leaf vertices, the root node also computes the IDs of the intermediate vertices in the Neuron tree, which are called the forwarding vertices. The forwarding vertex at the \( i \)-th level and \( m \)-th position in a Neuron tree is denoted \( H_{im} \), and its ID is obtained as follows.

\[
H_{im} = \begin{cases} 
H(e_1) & \text{if } i = 0 \\
\frac{H(e_{i+1}) + H(e_{i+1}(m+1))}{2} & \text{if } i \geq 1
\end{cases}
\]

where \( H_{01} \) is the ID of the root vertex, \( H_{0j} = H(e_j), \)

\[ H_{ij} = H(e_2), ..., H_{ij} = H(e_{ij+1}) \]

are the leaf vertices, and \( 0 \leq i \leq n, 1 \leq m \leq 2^{i-1} \). The forwarding vertex may be mapped to a Tornado node, which is referred to as a forwarding node.

3.2.2. Service Access Algorithms

Service registration and publishing. Figure 3 presents the algorithm for service providers to register a service to Neuron. First, the hashed values of the expression of the given service are sorted (see Line 4). Then the service object is stored to the home node with the sorted and concatenated hashing keys (see Line 5). The sorting operation guarantees that there is an identical order of hashed values for a set of expressions. Note that the publish operation is provided by Tornado, which routes and stores the designated data items to a Tornado node with the ID numerically closest to the specified hashing value.

To publish the service, the home node then computes the IDs of the forwarding vertices via Equation 4. The forwarding vertices are mapped to Tornado nodes, and the Neuron tree is constructed by setting up the output and input channels between the nodes. The ID of the input/output channel is again a hashed value rather than the stationary IP address of the node. This allows the use of the fault-resilient routing method of Tornado and consequently leverages Neuron’s reliability. In addition, to boost performance, it is possible to take a shortcut route between two nodes by employing the directories in Tornado.
(1) Input: the service expression \((e_1, e_2, e_3, \ldots, e_n)\);
(2) Output: a set of discovered services, \(D\);
(3) Procedure Discovery begin
(4) \(D = \emptyset\);
(5) Let \(o_1, o_2, o_3, \ldots, o_n\) be the sorted list of the hashed values of the service expression \(e_1, e_2, e_3, \ldots, e_n\);
(6) Let \(leaf = H(o_1 \oplus o_2 \oplus o_3 \oplus \cdots \oplus o_n)\);
(7) For each tag \(t \leftarrow Tag(node_{root})\)
(8) For each hashed value \(v \in IC(leaf, t)\)
(9) \(D = D \cup ReachRoot(v, t)\);
(10) Return \(D\);
(11) End.

Figure 4. The service discovery algorithm that looks up services conforming to the given expression \(e_1, e_2, e_3, \ldots, e_n\), where Tag \(v\) contains the tags in a given virtual node with ID \(x\), and node \(v\) denotes a physical Tornado node with the ID closest to \(x\).

If a Neuron tree is heavily loaded, the cost to build the tree is at most \((n + 1) \times \log N\) hops of communication without overloading a particular forwarding node in such a Neuron tree. This includes the communication for routing a registration message towards its root node and that for publishing the service down to the nodes storing the leaf vertices, where \(n\) is the number of tuples in the expression of the service, \(N\) is the number of active nodes in Neuron and \(\log N\) is the routing overhead in Tornado. However, if the system is lightly loaded, the communication cost may be reduced to \(2 \times \log N\) hops. The detail is presented in Section 3.3 and [4].

Discovery. To discover a serve, clients issue a free form expression, and all the registered services conforming to the expression are returned. Conceptually, the discovery is performed by backtracking the Neuron trees from a node storing the leaf vertex conforming to the expression. The backtracking is terminated if it reaches the root node (i.e., the service home node). The root node then replies necessary information (e.g., the interface to the object) to the client. Note that it is possible to discover several services satisfying the request. In this case, the backtracking will visit the root nodes of several Neuron trees.

The discovery algorithm is presented in Figure 4. Again, the service expression is sorted first according to the hashed values of the expression. Then the requesting node sends the request message to the node with an ID closest to the hashing address obtained from the concatenation of the hashed values of the expression (see Line 6 in Figure 4(a)). This node stores the leaf vertex conforming to the specified expression. That leaf node then informs the node whose ID is closest to the received hashing address to perform the ReachRoot operation to backtrack to the root (see Figure 4(b)). Since services may share common attribute-value pairs, their Neuron trees have common leaf vertices. Thus, the leaf node may need to inform several nodes to do the backtracking. Discovery operation takes \(n \times \log N\) hops at most in a heavily loaded Neuron space.

Subscription. Once a client discovers a particular service, it may subscribe to and access the service. The subscription operation in Neuron is shown in Figure 5.

A subscription request is handled by associating the subscription to the leaf vertex of the Neuron tree representing the subscribed service. The ID of the leaf vertex is the hashing key of the expression specified by the client. Again, since multiple Neuron trees may share common leaf vertices, a subscription request may subscribe to several services at once. Note that, a client may not be a Tornado node. For example, it may be a thin-client device. The set \(\text{Sub}(\nu)\) in Figure 5 is used to handle this problem by recording the network addresses (e.g., the IP addresses) of the clients that subscribe to the service with the hashing address \(\nu\).

(1) Input: the ID of the home node, \(\text{home}\), and the states to be updated;
(2) Procedure Refresh begin
(3) For each \(\nu \in OC(\text{home}, \text{home})\) in \(\text{node}_{\text{home}}\)
(4) Update the states in \(\text{node}_{\text{home}}\);
(5) RefreshChild(\(\nu, \text{home}\));
(6) End.

Figure 6. The renewal algorithm, where (a) is performed in the root node and (b) is performed in forwarding/leaf nodes.
sible to further incorporate the transport protocol used (e.g., HTTP and FTP) in Sub(v). Since Neuron is based on the Tornado infrastructure, the communication cost of the subscription operation is \( |\log N| \) hops at most.

Renewal. Services need to renew their states to Neuron periodically or aperiodically, depending on the implementation of the services. A service will be removed quietly from Neuron if the associated time contract is expired. To renew the states, a service needs to send its up-to-date states to its home node. The home node then forwards the states to nearby forwarding nodes. The forwarding nodes further transmit the states down the Neuron tree until all the leaf nodes and associated subscribers are updated. Note that the renewal operation takes \( n \times |\log N| \) hops at most for a heavily loaded Neuron space. Figure 6 shows the algorithm.

3.3. Optimization and Management

The hash-based naming scheme in Neuron can evenly distribute the load under normal conditions, i.e., the underlying Tornado nodes will store similar amount of service objects. However, some services may be more popular than others and thus are frequently accessed. To handle the increasing system load, Neuron implements an adaptive tree management strategy to recruit unused Tornado nodes. Of course, the load of a particular service may also be decreased. Due to the space limitation, we omit the descriptions for the tree management algorithm. The details can be found in [4].

3.4. Reliability

Since nodes in Neuron may fail and communication links may be broken, Neuron relies on the fault-resilient routing supported from Tornado. This is helped with the hash-based naming, which allows a Neuron vertex to delay the binding of the actual network address (e.g., the IP address) to the name of the overlaying node.

Suppose, for a given service tagged with \( t \), a node \( c \) detects that its parent or child node failed. The node will send a recovery request to a Tornado node with an ID closest to the failed parent or child node. Let us assume that it is the parent node of node \( c \) that failed, and the contacted Tornado node is \( p \). Node \( c \) will send a message containing \( \text{root}(c,t) \) to \( p \). Other nodes that detect the failure of the parent node will do the same. Eventually, \( p \) will receive all the recovery requests from the child nodes of the failed node. Similarly, the parent node \( p' \) of the failed node will also detect the failure and send a recovery request with \( l \in \text{leaf}(p',t) \) to \( p \), where \( \text{root}(p,t) \in \text{OC}(l,t) \). With this information, \( p \) can construct the virtual Neuron sub-tree it needs to host.

Reliability of Neuron can be further improved with the \( k \)-replication mechanism of Tornado, which replicates \( k \) identical copies to \( k \) Tornado nodes whose IDs are closest to the home node. It follows that each Neuron space is implicitly replicated to \( k \) spaces that are deployed over different Tornado nodes. When a Neuron node fails, the contents stored in the failed node can be recovered from the replica nodes. Since the replica node already has the most up-to-date states of the failed node, there is no need to reconstruct the Neuron sub-tree.

3.5. Scalability

Although Neuron can adapt to the changing load by varying the mapping of virtual forwarding nodes to Tornado nodes, the size of a Neuron space for a given service is obviously limited by the number of attribute and value pairs. A further limitation occurs when the Neuron tree is fully loaded, i.e., each virtual forwarding node is mapped to a physical Tornado node, and every node is loaded with excessive requests. One workaround is to forward the requests to the replica nodes. Note that the number of replica \( k \) is given by the service provider. However, it can be changed dynamically. In this way, it is possible to guarantee high availability and at the same time ensure scalability.

Intuitively, Neuron introduces a large amount of Tornado nodes to accommodate a great amount of requests while maintaining efficient registration, lookup and states propagation without producing hot spots for those popular services. However, if a service is not popular in demand, the corresponding Neuron tree may be degenerated and consequently hosted by a few of Tornado nodes. The size of a Neuron tree is thus highly dependant on the load of the participated Tornado nodes. Thus the size of a Neuron tree does not exponentially grow with the number of attribute-value pairs. This feature not only helps reduce the number of network hops required to access a particular service but minimize the resources used.

4. Performance Evaluation

We evaluated Neuron via simulation. For the P2P data routing, locating and storage infrastructure, we simulated 125 Tornado nodes initially. Each Tornado node is associated with a randomly and uniformly generated hashing key. All the Tornado nodes are equipped with a routing and a neighbor table. Each routing level in the routing table has two leader nodes, each taking care of a particular region in the node addressing space. Each neighbor table has four numerically closest active nodes. Although Tornado provides caching and directory map for performance optimization, we did not turn on these features in order to concentrate on the performance of Neuron.

Due to a lack of traces for service discovery protocols, we resorted to the simulation of a service object containing 4096 attribute and value pairs. The resultant Neuron tree has 8191 Neuron nodes. The hashing keys of the service object and the associated attribute-value pairs are generated randomly and uniformly. The primary performance metric is the path length, i.e., the number of hops required for routing a given request.

4.1. Scalability

We changed the network size dynamically from 125 Tornado nodes to 150000. There are total 150000 time steps simulated and each simulated node performs at most 10 periodical updates to the associated routing and neighbor tables. In each iteration, a new node is introduced. The measured path length is averaged over routes between randomly generated clients and service home nodes.
Figure 7. The measured path length versus the network size for (a) service registration and (b) subscription.

4.2. Stressing

In this experiment, we perform the stressing test by introducing random traffic incrementally. A client with a randomly generated ID is chosen to send a request to a randomly chosen Neuron leaf node in the simulated Neuron tree. The Neuron space is deployed on a Tornado network with 150000 nodes. Each Neuron node is assumed to accommodate only one request at a simulated time step. The requests are doubted continuously with the simulated time steps. Note that the Neuron tree will adapt to the system traffic by spreading the load to more Tornado nodes. The simulation terminated when the Neuron tree grew to a complete binary tree consisting of 8191 Neuron nodes. The performance for lookup and renewal is measured for varying sizes of the Neuron tree.

Figure 8 presents the path length for service lookup and renewal. We can see that the lookup performance scales logarithmically with the number of nodes in the Neuron space. The depth of the Neuron tree is obviously dominated by the number of attribute-value pairs. The overhead can be optimized by utilizing the directory provided by Tornado to route the messages with shortcuts. This optimization is not evaluated in this study though.

Similarly, the renewal performance also scales logarithmically. This means that additional hops are required to epidemically distribute the up-to-date states from the service object to the root node, plus the hops for distributing the states from the leaf nodes to the subscribers. Figure 8(b) depicts the number of hops required to transmit the states in the longest path of the simulated Neuron tree.

4.3. Fault-Resilience

To explore the robustness of Neuron in providing service availability, Neuron nodes simulated are randomly detached from the Neuron space. Initially, the simulated Neuron space consists of 64000 nodes. Figure 9 presents the service availability with a varying number of replicas for each Neuron node. The replicas will contain the interface of the service object, the internal Neuron tree data structure, and the addresses of the subscribers, etc.

We can see that when 50% of the nodes are failed, up to 80% of randomly generated requests can be served by the Neuron space with replication. When there are three repli-
cas, up to 95% of requests can be served. The percentage improves to 99% when there are seven replicas. Even active Neuron nodes are below 10%, the Neuron space can still serve 45%, 29% and 19% of requests with 7-, 3- and 1-replications.

5. Related Works

Sun Jini [6] provides a framework for discovering federated network devices via Remote Message Invocation (RMI). In Jini, services register themselves to the lookup servers, and clients discover desired services by querying the lookup server. Jini adopts the client-server model and thus may not be able to handle large numbers of requests with a single lookup server. Thus, Jini cannot be scaled up for supporting Internet-scale services. The lookup server in Jini may become a single point of failure, which is not adequate for dynamic and distributed environments. Another product similar to Jini can be found in [9]. Neuron adopts the peer-to-peer model rather than the client-server model, which is appealing for providing wide-area services without suffering the single point of failure.

Another representative service discovery protocol is SLP [10] proposed by IETF, which employs centralized directory agents. Similar to Jini, SLP may have performance bottleneck and single point of failure with the centralized approach. Although the directory agents of SLP can be configured via the standard directory protocols (e.g., LDAP), deploying the directory servers is cumbersome. In addition, partitioning the naming space to the directory agents requires extra overhead. This further complicates the system deployment. In contrast, Neuron does not rely on particular servers, and the Neuron nodes cooperate to serve requests.

The Berkeley SDS [2] extends SLP to support secure and authenticated communication with a static hierarchical structure for wide-area services. The hierarchical architecture cannot self-configure to adapt to the changes of distributed environments. SDS shares some common features of SLP and cannot tolerate failure of nodes and communication links. It also needs to partition the naming space.

The work most related to Neuron is perhaps INS [1]. INS is equipped with a self-organizing resolver network, although the resolver network may be highly sensitive to the failure of nodes and communication links. Services in INS are expressed in a hierarchical fashion and each resolver maintains an identical copy of such a hierarchical tree representation. The results are that (1) the hierarchical presentation for services limits the expressiveness of services, and (2) aggregating the tree presentation for a service into a resolver node constraining INS to support small-area services only. Although it is possible to scale INS to support wide-area services by introducing domain space resolvers, this introduces administrative overheads. Neuron is quite different from INS as Neuron is implemented on top of Tornado, which provides highly reliable and scalable Neuron space for service discovery. Naming in Neuron is relatively free and poses no hierarchical relationship. This gives higher expressiveness. More importantly, the naming space is distributed to the Neuron nodes to share the load.

Jini [6]'s accompany product JXTA search [7] is based on the P2P model although it basically adopts a centralized approach. JXTA search allows a distributed design by organizing JXTA search hubs. The protocol for the distributed JXTA search, however, is left for application programmers.

6. Conclusions

In this paper, a wide-area service discovery infrastructure called Neuron is introduced. Neuron is based on the P2P model and implemented on top of Tornado, a P2P data routing, locating and storing substrate. Neuron is designed for aggregating the resources in a wide-area network, and it becomes more powerful if more resources can be exploited. Neuron can tolerate node or link failure and disconnection, while maintaining high service availability. This helps ensure the durability of registered services. Neuron can adapt to the changing load by distributing the load to the participating nodes. Clients can subscribe to the services of interest and receive their up-to-date states. The services can be described in free forms. Our simulation study shows that Neuron scales logarithmically with the number of active nodes. The cost to perform service registration, discovery, subscription and renewal takes O(logN) hops at most.

References


Figure 9. The service availability versus the number of remaining active nodes, where there are 0, 1, 3 and 7 replicas for each Neuron node.