Integrating heterogeneous information services using JNDI

Dirk Gorissen, Piotr Wendykier, Dawid Kurzyniec, and Vaidy Sunderam

Emory University
Dept. of Math and Computer Science
Atlanta, GA 30322 USA
dgorissen@gmail.com, {wendyk,dawidk,vss}@mathcs.emory.edu

Abstract

The capability to announce and discover resources is a foundation for heterogeneous computing systems. Independent projects have adopted custom implementations of information services, which are not interoperable and induce substantial maintenance costs. In this paper, we propose an alternative methodology. We suggest that it is possible to reuse existing naming service deployments and combine them into complex, scalable, hierarchical, distributed federations, by using appropriate client-side integration middleware that unifies service access and hides heterogeneity behind a common API. We investigate a JNDI-based approach, and describe in detail two newly implemented JNDI service providers, which enable unified access to 1) Jini lookup services, and 2) Harness Distributed Naming Services. We claim that these two technologies, along with others already accessible through JNDI such as e.g. DNS and LDAP, offer features suitable for use in hierarchical heterogeneous information systems.

1. Introduction

Resource information, registration, and discovery services are of crucial importance in heterogeneous distributed systems, as they provide the necessary bridge between resource providers and consumers. Existing information services, targeting vastly different systems and addressing diverse requirements, vary greatly in the update and query capabilities, levels of security and fault tolerance, genericity, scalability, and responsiveness. For instance, DNS provides a name resolution service which scales world-wide but is specialized, lacks strong consistency, and has limited query capabilities. These features make it suitable for managing simple textual data collections for which updates are rare and sophisticated queries are unnecessary. On the other hand, information services targeting more dynamic and diversified data sets, such as Jini (with its leasing and event notification mechanisms), are usually less scalable. Often, a hybrid approach is required to balance these conflicting objectives. Independent projects have developed their own solutions, typically involving complex, distributed, hierarchical information services requiring custom installation, configuration, and maintenance, thus adding to the existing burden of IT software support. Moreover, these specific components are usually not interoperable, which can be a major obstacle to building heterogeneous systems. In this paper, we argue that hybrid, large-scale, distributed, hierarchical information services can be constructed by assembling existing, off-the-shelf, heterogeneous software components. We claim that access homogeneity can be achieved at the client-side by using an appropriate integration technology, such as Java Naming and Directory Interface (JNDI). We demonstrate how the JNDI model can be used to integrate lookup and discovery mechanisms, such as Jini, HDNS, LDAP, DNS, and others, to build complex, scalable, and powerful information services, while potentially reusing existing middleware deployments, thus promoting evolutionary development of the IT infrastructure and facilitating cross-site integration.

2. Related Work

The choice of an information service implementation depends on a number of factors, including: expected scope (i.e. number and type of relationships between collaborating parties), application requirements (e.g. simple device sharing vs. parallel job submission), existing infrastructure and deployment environment (e.g. firewalls and administrative domains), expertise of the IT personnel, or even the personal preference. Not surprisingly then, different projects have adapted different solutions to address their particular requirements. Below we list some of the existing grid projects together with the approach they have chosen to implement service discovery and registration. Subsequently, we give examples of integration efforts that, like ours, try to tackle the limitations of these projects.

ICENI: The Imperial College e-Science Networked Infrastructure (ICENI) is a mature, service-oriented, integrated grid middleware built around Java and Jini. Its resource discovery and registration tasks are handled mainly through Jini, augmented with higher level capabilities such as XPath queries and semantic matching. Though predominantly Jini based, it has been demonstrated [13] that ICENI's service-oriented architecture can also be ported to JXTA [17] and OGSA (albeit with some restrictions).

JGrid [21], JISGA [30] and ALiCE [26]: these projects are similar in that all use the Jini framework for resource information management: Resources are represented by Jini services that are stored in the Jini lookup service (LUS). The LUS supports service interface and attribute based matching. In addition JISGA provides an OGSA-compliant interface, and JGrid has extended Jini model with a wide-area discovery mechanism.

Triana [25]: Created at PPARC as part of the GridOneD project the Triana aims to be a pluggable grid middleware implemented as a Grid Application Toolkit (GAT) which provides a portable set of core services. Resource registration and discovery component of the Triana GAT provides bindings to JXTA and OGSA (experimental).

Globe [3]: The Globe middleware platform is designed to enable the flexible development of global-scale Internet applications. It is based on a distributed shared objects model from which applications are built. These distributed shared objects are automatically replicated across the network and they are registered in a two-tier naming service. At the virtual level, each object is assigned a human readable, globally unique name, stored in DNS. To maintain the binding between an object replica and its current location (IP address), a separate, hierarchical *on location service* is used. Object replica servers are located using the Globe Infrastructure Directory Service which is a hierarchy of LDAP servers.

As is apparent in these examples, grid information services are characterized with substantial complexity and large heterogeneity. The complexity places significant burden on resource administrators, who are responsible for configuring and maintaining the middleware. The heterogeneity, on the other hand, hampers interoperability between grid deployments, and complicates application development. We suggest that it is possible to overcome those issues via an integration middleware that would enable interoperability between service implementations and hiding service heterogeneity behind unified, semi-transparent APIs, providing uniform access to common capabilities while also permitting use of custom, service-specific functionality.

This idea of inserting an extra layer of abstraction onto which multiple information services are mapped is not new. Other projects that have taken up this hourglass model include:

MonALISA [14]: The goal of the MonALISA framework is to provide service information from large, distributed and heterogeneous systems to a set of loosely coupled higher level services in a flexible, self describing way. These higher level services can then be queried by others and are implemented as Jini services that fully leverage the Jini framework (security, ServiceUI, leasing, ...). For non Jini clients a webservice bridge is available. MonALISA is already in a mature state of development counting a large number of features such as support for SNMP and integration with external monitoring tools like Ganglia.

Globus MDSv4 [12]: standard Web Service interfaces form the core abstraction of MDS. Globus currently uses a WSRF-based Monitoring and Discovery System (MDS) to provide information about the current availability and capability of resources. Service data is generated by Service Data Provider programs which are then aggregated into Grid Service instances, each of which provides information through Service Data Elements (SDE). A MDS Index service allows for aggregation capabilities. Data services have their own special registry called a Grid Data Service Registry (GDSR). Previous versions of Globus and MDS, which are still in active use, relied on LDAP-based registries [7, 8].

R-GMA [11]: The Relational Grid Monitoring Architecture provides a service for information, monitoring and logging in a heterogeneous distributed computing environment. R-GMA makes all the information appear like one large Relational Database, "the hourglass neck", that may be queried to find the information required. It consists of independent Producers which publish information into R-GMA, and Consumers which subscribe. R-GMA uses SOAP messaging over https for all high-level user-to-service and service-to-service communications. R-GMA was originally developed under the European DataGrid project but is now part of the wider European EGEE project.

Hawkeye [2]: Hawkeye was born from the Condor project and is a custom monitoring and management tool for distributed systems. Hawkeye adopts pluggable architecture, different modules that provide extra functionality (such as monitoring available disk space) can be easily plugged in. Though it retains a strong bias towards Condor it can be used to monitor and integrate different distributed systems.

While the above mentioned systems provide adequate levels functionality to support their target application areas, they are not interoperable, and they impose substantial setup and maintenance efforts. In contrast, the approach exploited in this paper emphasizes reuse of existing information service deployments, and offers a possibility to aggregate different, heterogeneous, distributed information services into a single name space that is available to clients via an unified API.

3. JNDI

The JNDI is an application programming interface (API) that provides naming and directory functionality to applications written using the Java programming language. It is designed to be independent of any specific directory service implementation, allowing variety of directory systems (including new, emerging, and already deployed) to be accessed in a common way. The JNDI architecture consists of a client API and a service provider interface (SPI). Java applications use the JNDI API to access a variety of naming and directory services, while the SPI enables pluggability of directory service implementations [23]. The JNDI architecture is illustrated in Figure 1.

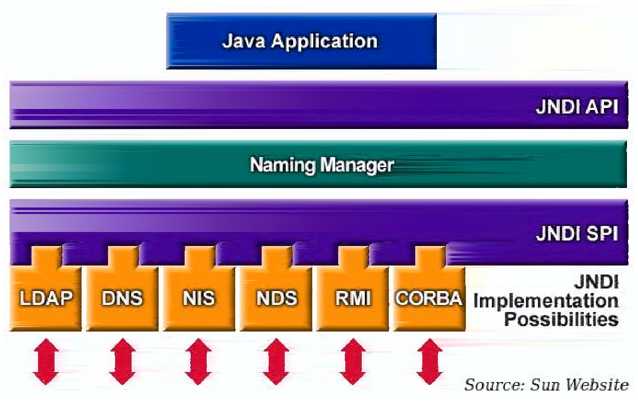

Figure 1. The JNDI Architecture

Currently, JNDI is used mostly in Enterprise Java (J2EE) containers to manage information related to Enterprise Java Beans (EJB). In the grid computing context, virtually all JNDI use cases are related to accessing LDAP directories (e.g. MDS used in Globus v2) or, in some cases, COS naming servers [29]. The JNDI has also been used in agent frameworks [6]. Specialized SPIs have been developed for the smart card-based Personal Naming and Directory Service [19] and the naming service used in the DREAM distributed services platform [22].

JNDI exhibits a number of characteristics that make it suitable for heterogeneous information services. First, it features a simple and generic "lowest-common-denominator" base directory interface, so it can support a wide range of service providers. Moreover, JNDI provides uniform but flexible APIs for lookup queries and metadata management. Finally, JNDI provides support for linking multiple, heterogeneous naming services into a single, aggregated name space (federation), with a simple URL-based naming scheme introduced to identify data entries.

Obviously, a lowest-common-denominator API unifying vastly different services would suffer from over-simplification if it was truly flat and opaque. Therefore, JNDI defines a hierarchy of interfaces, leaving the provider the choice of the supported conformance level. The API is designed to balance the genericity with flexibility. Data entries are represented as <*name*, *object*, *attributes*> tuples. Depending on the underlying service provider implementation, the supported "object" and "attributes" types may be arbitrarily complex; the JNDI specification allows for flexibility while recommending certain minimum conformance levels that every provider should satisfy (e.g. ability to bind any serializable object). Further, JNDI provides APIs for attribute-based queries. Although the syntax of such queries is mandated (and biased towards LDAP), extensions are possible.

The unavoidable tension between genericity and flexibility results in certain tradeoffs. Some specialized capabilities of backend services cannot be easily represented via the API; an example includes the leasing functionality available in Jini. At the same time, JNDI does not fully hide backend service heterogeneity – for instance, clients are sometimes required to supply service-specific configuration parameters or credentials. Finally, since JNDI is a Java technology, the client libraries are not directly accessible from native languages, for which analogous APIs have not been (yet) implemented. Notwithstanding these disadvantages, we feel that they are outweighed by the advantages and can be countered. In particular, we note that access unification can often be achieved to a larger extent than immediately apparent, since certain capabilities missing at the server side can be emulated at the client side. Also, the service-specific configuration stages can usually be isolated from the application layer.

4. HDNS

4.1. Overview

HDNS [27] is a fault-tolerant, persistent, distributed naming service initially developed for the Harness project [20]. While developing JNDI Service Providers, a completely new version of HDNS (based on JGroups [4]) has been designed and implemented. HDNS establishes a group of naming service nodes which maintain consistent replicas of the registration data. Read requests can be handled entirely by any of the nodes, which facilitates load balancing and reduces access latencies. Write requests, in turn, are propagated to each member of the group, enforcing consistency of the replicas. Each node maintains persistent view of the registration data on a local disk, synchronized in fixed time intervals and upon process exit. The service can thus recover the state after a complete shutdown/restart. To support recovery from partial failures, mechanisms are provided that enable individual crashed/restarted nodes to re-join the group. Furthermore, the service is capable of recovering from network partitions and re-synchronizing the distributed state. These characteristics make HDNS an adequate choice in situations where read requests are predominant but fast update propagation and high failure resilience is required.

4.2. JGroups

JGroups is a toolkit for reliable multicast group communication. It can be used to create groups of processes whose members can send messages to each other. Members can join and leave the group at any time; the toolkit detects membership changes and propagates them to participants. The most powerful feature of JGroups is a configurable protocol stack, allowing to defer quality-of-service decisions regarding fault tolerance and scalability until run time. For instance, the Virtual Synchrony protocol suite guarantees an atomic broadcast and delivery. However, it comes at the cost of scalability - the entire group is only as fast as its slowest member. An alternative protocol suite uses Bimodal Multicast [5], which improves scalability, for the price of probabilistic message delivery reliability. The latter suite was chosen as the default in HDNS.

4.3. Design choices

HDNS has been implemented as a relatively thin software library, based on existing, well-established underlying technologies - JGroups as a communication substrate and H2O [18, 24, 1] as a hosting environment. Consequently, HDNS exhibits many advanced features inherited from mentioned projects. Owing to dynamic deployment features of H2O, HDNS service can be dynamically deployed on participating nodes from a remote network repository, which is particularly useful during the update or maintenance process. Furthermore, H2O provides HDNS with security infrastructure, allowing to control access via user-defined security policies and featuring flexible authentication technologies. Finally, HDNS uses distributed event notification mechanism offered by H2O to implement the JNDI event notification functionality. Similarly, distributed communication and synchronization of replicas is achieved by using appropriate mechanisms of the JGroups toolkit. However, in order to fully support recovery from network partitioning, we needed to implement an additional protocol for the JGroups protocol stack. After a transient network partition, the PRIMARY_PARTITION protocol resolves state conflicts by uniquely selecting the partition deemed to have the valid state, and forcing other partitions to re-synchronize.

5. JNDI Service Providers

The pluggable architecture and genericity of JNDI, coupled with logical and well-documented APIs, enables relatively easy implementation of custom service providers. This section contains a description of two new service providers: Jini-based and HDNS-based, the choice of which was motivated in part by their potential usefulness in building hierarchical information systems.

5.1. The Jini Provider

Jini [16, 9, 28] is a technology developed by Sun Microsystems, providing an environment for building scalable, self-healing and flexible distributed resource sharing systems. Jini enables Java Dynamic Networking, where clients bind dynamically to any network service that is capable of providing needed functionality, regardless of the name, location, wire protocol, or other implementation details of the particular service instance [9].

Mapping of Jini functionality onto the JNDI abstractions has proven to be a non-trivial task, due to differences in philosophy between the two technologies. In particular, problems arise due to the narrower focus of Jini which is specifically designed to manage *services*, characterized by capabilities expressed by Java remote interfaces, and not as a general purpose information storage facility. Below we list specific encountered problems and their solutions.

State and Object Factories. To be able to store generic name-value mappings in the Jini registry, a translation mechanism is needed to convert them into fake Jini service stubs upon registration, and back to the original form upon retrieval. We have accomplished this by using the JNDI abstractions of object and state factories, which perform the necessary translations automatically when the appropriate API methods are invoked.

Handling leases. To avoid stale references, Jini adopts a notion of leasing [15]. Validity of a data entry in a naming service has to be continuously renewed, typically by the service itself, or else it expires and the entry is removed. No such abstraction exists in the JNDI API, which does not specify any explicit data expiration policy. Hence, Jini leases cannot be easily passed upward to the application layer through the JNDI API. Therefore, we decided to handle Jini leases entirely in the service provider implementation layer: the provider automatically renews leases of all entries that it has previously bound, until they are explicitly removed, or until the Java VM exits.

Atomicity and consistency. One of the biggest and most unexpected Jini-JNDI mapping issues emerges from the implementation of a *bind* primitive. The JNDI *bind* has atomic semantics: the operation should succeed if there was no entry bound to the specified name, and otherwise fail throwing an exception. Unfortunately, the Jini LUS does not provide any such primitive that could be used to implement this functionality strictly; aiming at achieving idempotency, Jini registration methods always overwrite the previous value. Hence, in order to ensure atomic consistency in concurrent setups, resort to distributed locking was needed. This forced us to adopt Eisenberg and McGuire's algorithm [10], which depends only on the basic read and write primitives, but which is rather costly: it takes 3 reads and 5 writes to enter and leave a critical section in the uncontended case. This means that our strictly compliant implementation of JNDI *bind* on top of Jini induces at least an eight-fold increase in latency in comparison to the basic Jini primitive, which may or may not be a problem, depending on whether the application is latency-bound. We note however, that in many applications, binding/unbinding of a given named entry is performed only by a single writer (or the "owner"), in which case the semantics of *bind* can be safely relaxed. We thus provide applications with an option to disable the strict semantics, removing the performance penalty by sacrificing the atomicity.

5.2. HDNS Provider

The control over the source code of HDNS allowed us to avoid certain problems encountered in the context of Jini. HDNS was designed in a way that mapping through JNDI was simple. As a result, a distributed locking algorithm was not needed to implement an atomic *bind* for HDNS. In fact, all methods from JNDI `DirContext` interface are atomic in the HDNS service provider. Besides this difference, there is a close affinity between two described providers. HDNS, analogous to Jini utilizes the concept of object and state factories, both of the providers also have a very similar mechanism for handling leases.

6. Federation

Federation is the process of "hooking" together multiple, distributed, and possibly heterogeneous naming systems so that the composite behaves as a single, possibly hierarchical, aggregate naming service. In JNDI, resources in a naming service federation are identified by composite URL names, which can span multiple substrate naming services [23]. For instance, the URL:

```
ldap://host.domain/n=jiniServer/jxtaGroup/myObject
```

links together LDAP, Jini and JXTA services. From the end-user's perspective, accessing an object through federation is fully transparent, and reduces to passing the composite URL to the appropriate API function:

```
Object val = new InitialDirContext().lookup(
"ldap://host/n=jiniServer/jxtaGroup/name");
```

Behind the scenes, JNDI parses the URL, recognizes it as an LDAP reference, and passes it to the LDAP service provider. That provider, in turn, performs lookup on the string "n=jiniServer", and retrieves a value that is, in this case, interpreted as a reference to the Jini service previously bound to the LDAP service. The request is thus further propagated to the Jini SPI, which performs a lookup on the "jxtaGroup" segment, and identifies it as a reference to the JXTA naming service. Finally, the request is delegated to the JXTA service provider, which retrieves the target object data bound to "name".

From the API perspective, linking naming services into federation is straightforward, and reduces to binding the context interface of one naming service to another naming service:

```
DirContext intCxt = new InitialDirContext();
DirContext jiniCxt =
initCxt.lookup("jini://host1");
DirContext hdnsCtxt =
initCxt.lookup("hdns://host2");
hdnsCxt.bind("jiniCxt", jiniCxt);
// now, the URL: hdns://host2/jiniCxt refers
// to the embedded Jini directory
```

We have implemented the necessary programmatic support for federations in both providers described in this paper, allowing them to be federated with information services such as DNS, LDAP, or a local filesystem storage, for which the appropriate JNDI providers already exist.

In a wide-scale, hierarchical, distributed information service, the requests originate at the root naming service and are propagated downwards, to be eventually handled by the "leaf" information services. The root naming service thus receives a massive load of requests, necessitating extremely scalable approaches, with DNS being a good candidate. On the other hand, the "leaf" naming services (e.g. LDAP servers or Jini registries at the department level) can be expected to encounter frequent updates and computationally intensive queries, and require rapid state propagation and event notifications. Suitable technologies for both those extreme cases already exist. The biggest research challenge is related to the *intermediate* layer, since it has to balance both: (1) distribution and high scalability requirements, to ensure high system throughput and minimize latencies by matching requesters to local nodes, and (2) rapid propagation of updates, with the capability to handle moderate to high update frequencies. We argue that HDNS can be a viable technology to satisfy these requirements. On one hand, state replication allows it to achieve high scalability, whereas its error recovery mechanisms ensure high failure resilience. On the other hand, its state synchronization methodology ensures fast update propagation with configurable consistency levels (examples including bimodal multicast or virtual synchrony).

We thus envision the following, JNDI-based methodology for composing individual, department-level, existing and operational information services, into larger, aggregated name spaces. We propose that a collection of HDNS nodes is deployed into the distributed system, so that a HDNS node can be found in the proximity of any large group of users. Additional nodes can be deployed dynamically at a later stage as well, while the system is already in operation. The replicated information shared by all HDNS nodes is the set of references to all department-level naming services (e.g. LDAP servers or Jini registries) present in the entire composite system. Since deployment or discontinuation of an information service is a rare occurrence, we expect relatively small update frequency at the HDNS level, unless the managed network is very substantial in size. Finally, in order to hide the aspects of distribution, we propose to anchor the federated naming system in DNS, so that a common, well-known service name is resolved to a nearest HDNS node. For instance, when querying the status of an object referred to by the URL "dns://global/emory/mathcs/dcl/mokey", JNDI client would contact DNS to find the address of a nearest HDNS node belonging to the "global" federation, then it would use HDNS to query for the address of the "emory/mathcs/dcl" LDAP server, and finally, it would issue the "mokey" object query to that LDAP server. In the next section, we present our preliminary experiments and small-scale stress tests, aiming to evaluate performance of HDNS and Jini providers alongside DNS and LDAP, and to assess viability of the proposed federation scenario.

7. Experimental evaluation

While adding an additional abstraction or adaptation layer may be elegant and appropriate from the design point of view, care must be taken that the performance penalty is not too large. To quantify the overhead introduced by the JNDI provider layer, we have run a preliminary series of benchmarks on a local Gigabit Ethernet network. The Jini LUS service, used in the benchmarks, has been running on a Pentium 4 2.4 GHz machine with Mandriva Linux 2005 and 1 GB of RAM. Similarly, the HDNS service has been installed on two identical dedicated machines. In addition to the Jini and HDNS tests, we have run the benchmarks for the DNS and LDAP JNDI providers, using the `Bind` and `OpenLDAP` servers installed on similar dedicated machines in the same LAN. In the throughput experiments, a single client machine (2.6 GHz Intel Celeron with 1 GB RAM, running Microsoft Windows XP) issues a series of requests from an increasing number of client threads (between 1 and 100). Each client thread issues consecutive requests (lookup requests for the read test and rebind requests for the write test) with 50 ms pauses between requests (i.e. with the frequency of up to 20 Hz). We measured the ability of the service to withstand the increasing load as a number of requests per second that have been successfully handled. In the ideal case (if the request processing time is negligible), the request frequency is 20 Hz per thread, and the number of completed operations per second is 20 times the number of client threads.

Figure 2 shows "lookup" results for standalone Jini, and for JNDI-Jini provider with (1) strict and (2) relaxed bind semantics. It can be seen that the standalone Jini can handle up to about 400 requests per second, and its throughput starts decreasing afterwards. The serialization layer introduced by the JNDI-Jini provider reduces the performance by about 25%, yielding the peek throughput of about 300 requests per second. The "strict" versus "relaxed" semantics, which apply only to write operations, did not affect the "lookup" results. The output for "rebind" is shown in Figure 3. The peek Jini write throughput has been measured at about 140 operations/s. The Jini-JNDI provider throughput approaches 80 operations/s for the relaxed semantics, and 20 operations/s for the strict semantics. This 7-fold decrease, caused by the need for extra communication, indicates that strict bind semantics should be disabled whenever possible, and otherwise a proxy-based solutions should be adapted so that the necessary locking is performed locally (near the Jini LUS, e.g. on the same host), exposing the atomic interface to the client.

Figure 4 presents "lookup" tests for HDNS and JNDI HDNS provider. In this test, all lookup requests are sent to the same HDNS node, so the results show a per-node throughput. As it can be seen, HDNS demonstrates excellent scalability; we have not been able to identify the peak throughput as it exceeds 1800 read operations per second. The HDNS JNDI provider layer does not introduce a noticeable overhead (due to the fact that HDNS server has been implemented with the JNDI support in mind). The "rebind" results for HDNS are shown in Figure 5. Write operations in HDNS impose a substantial overhead, due to the necessity to propagate the distributed state to all HDNS nodes. In our experiments, we have observed the peak write throughput at about 200 operations/s. The tests exposed an issue with the HDNS implementation: the service response time does not degrade gracefully as the number of clients increases; we observe a rapid throughput decline (instead of levelling off) for number of clients exceeding 20. We have traced the problem to the buffer management in the underlying JGroups implementation; flooding the server with requests cause internal JGroups message queues to grow without bounds, eventually causing memory exhaustion and server crash. We are currently investigating possible approaches to address this issue.

Figures 6 and 7 show throughput of `Bind` (DNS) and `OpenLDAP` services, respectively, when accessed via standard JNDI providers. As could be expected, DNS exhibits excellent scalability, with peak throughput per node exceeding 1800 lookup operations/s. Similarly, very good write throughput has been observed for the LDAP server. Surprisingly, the read throughput of `OpenLDAP` plateaus at about 800 operations per second, leaving server resources (CPU, network, memory) unsaturated. We are currently investigating causes of this phenomenon; we suspect that the anomaly is due to some automatic slowdown mechanism, such as a countermeasure against Denial-of-Service attacks.

These preliminary experiments lead us to conclude that the proposed HDNS service is a viable technology for implementing very scalable distributed lookup services, which – unlike DNS – support remote updates and ensure their rapid propagation. However, the implementation needs improvement to be able to gracefully handle update overload. Also, we note that `OpenLDAP` service exhibits excellent responsiveness to update requests, and is therefore feasible for management of dynamic data sets. The scalability of Jini scores somewhat lower; yet, in the environments with Jini already deployed, the JNDI interface can provide standardized access to it for the cost of some extra overhead.

Our preliminary tests indicate that the individual performance characteristics of the discussed JNDI providers are preserved when they are combined into a federated name space. We thus believe that HDNS is a promising technology for implementing intermediate layers in the large scale federated information service, given its excellent lookup scalability and fast update propagation. Further research and experimentation is required to validate feasibility of the proposed approach in large-scale network settings.

8. Conclusions

In this paper, we suggested a JNDI-based methodology of merging individual naming services into aggregate, hierarchical information systems. We described two new implementations of JNDI service providers, enabling seamless access to Jini lookup services, and to the fault tolerant Harness Distributed Naming Service. Extrapolating from these two cases, we argue that unification of the naming service access methodology is indeed possible and beneficial, despite the differences in design models and implementation strategies. We present our preliminary experimental results that suggest that HDNS may be a viable candidate for organizing isolated information service deployments into a DNS-anchored, aggregated name space. We believe that the approach enabling API standardization and

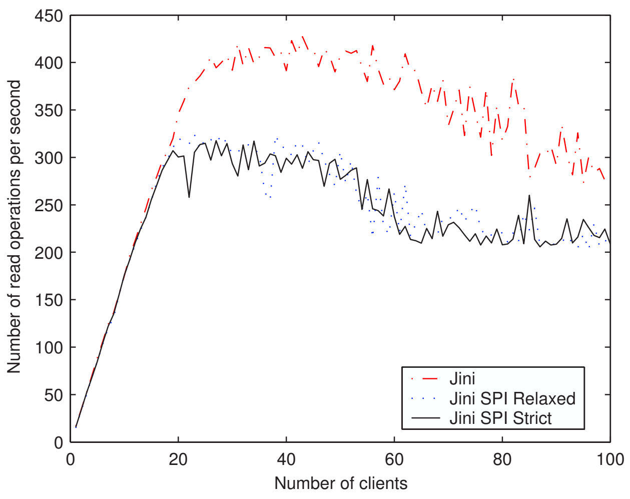

Figure 2. Throughput of Jini and JNDI Jini provider, lookup operations (read)

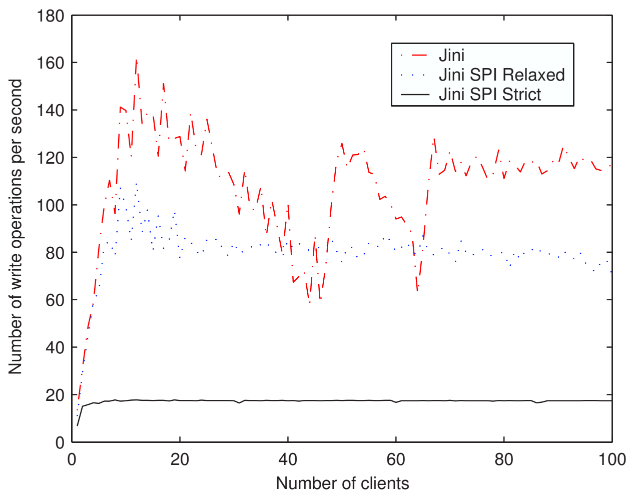

Figure 3. Throughput of Jini and JNDI Jini provider, rebind operations (write)

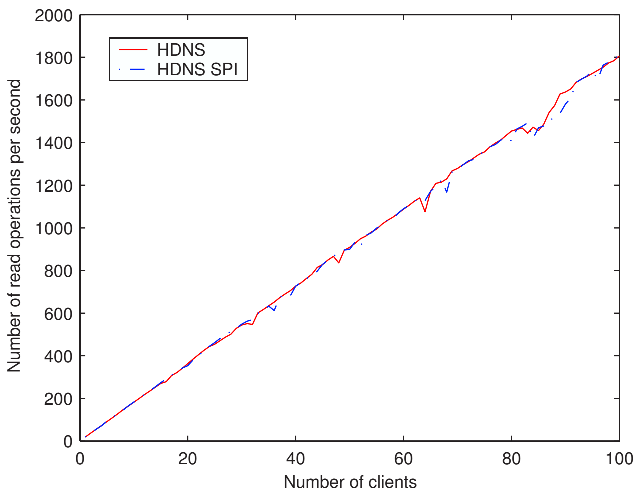

Figure 4. Throughput of HDNS and JNDI HDNS provider, lookup operations (read)

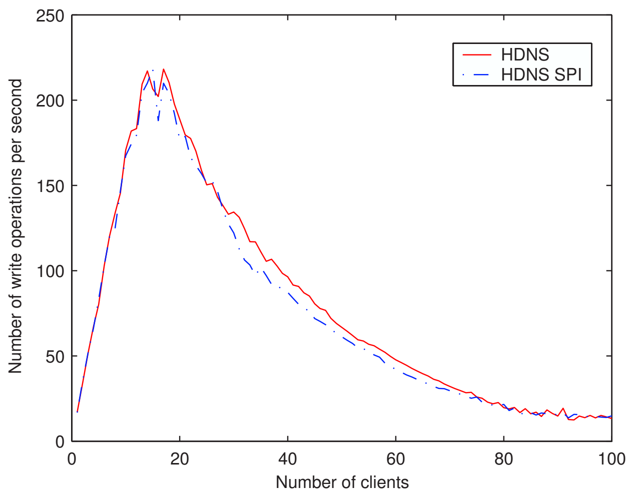

Figure 5. Throughput of HDNS and JNDI HDNS provider, rebind operations (write)

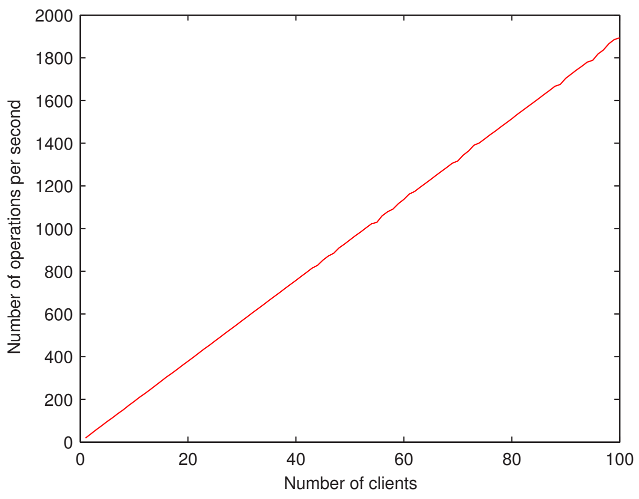

Figure 6. Throughput of JNDI-DNS, lookup operations (read)

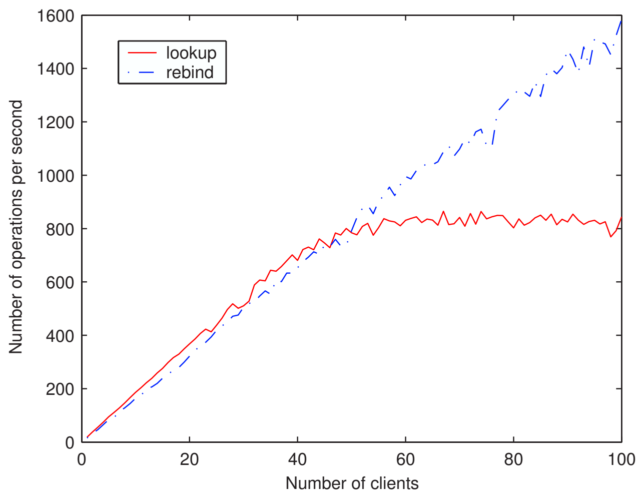

Figure 7. Throughput of JNDI-LDAP (OpenLDAP), read/write

seamless service federations, can result in a substantial decrease of deployment and maintenance costs, and in improved scalability and interoperability, without a significant sacrifice in functionality.

Even though our initial results are promising, further investigation and larger scale experiments are necessary to validate feasibility of the proposed approach in real-world settings. Building a large scale information service federation, and its thorough experimental evaluation, will therefore be the focus of our future work.

References

[1] Harness framework hompage. http://www.mathcs.emory.edu/dcl/harness.

[2] Hawkeye, a monitoring and management tool for distributed systems. http://www.cs.wisc.edu/condor/hawkeye/.

[3] A. Bakker, I. Kuz, M. van Steen, A. Tanenbaum, and P. Verkaik. Design and implementation of the globe middleware. Technical Report IR-CS-003, Vrije Universiteit Amsterdam, June 2003.

[4] B. Ban. Design and implementation of a reliable group communication toolkit for Java, 1998.

[5] K. Birman, M. Hayden, O. Ozkasap, M. Budiu, and Y. Minsky. Bimodal multicast. Technical Report 98-1665, Cornell University, 1998.

[6] K.-F. Blixt and R. Oberg. Software agent framework technology (SAFT). Master's thesis, Linkping University, 2000.

[7] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*. IEEE Press, August 2001.

[8] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.

[9] K. W. Edwards. *Core Jini*. Prentice Hall, 2 edition, 2001.

[10] M. A. Eisenberg and M. R. McGuire. Further comments on Dijkstra's concurrent programming control problem. *Communications of the ACM*, 15(11):999, November 1972.

[11] A. et al. The relational grid monitoring architecture: Mediating information about the grid. *Journal of Grid Computing*, 2(4), 2004.

[12] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, 2002.

[13] N. Furmento, J. Hau, W. Lee, S. Newhouse, and J. Darlington. Implementations of a service-oriented architecture on top of Jini, JXTA and OGSI. In *2nd European AcrossGrids Conference (AxGrids)*, volume 3165 of *LNCS*, pages 90–99, Nicosia, Cyprus, Jan 2004.

[14] P. G. R. V. H. B. Newman, I.C.Legrand and C. Cirstoiu. Monalisa : A distributed monitoring service architecture. In *Proceedings of the 2003 Computing in High Energy and Nuclear Physics*, 2003.

[15] P. Jain and M. Kircher. Leasing pattern. In *7th Pattern Languages of Programs Conference*, Allerton Park, Illinois, 2000.

[16] Jini homepage. `http://www.sun.com/jini/`.

[17] Project JXTA. `http://www.jxta.org`.

[18] D. Kurzyniec, T. Wrzosek, D. Drzewiecki, and V. Sunderam. Towards self-organizing distributed computing frameworks: The H2O approach. *Parallel Processing Letters*, 2(13):273–290, 2003.

[19] A. Macaire. An open terminal infrastructure for hosting personal services. *Technology of Object-Oriented Languages and Systems (TOOLS 33)*, 33:10–21, 2000.

[20] M. Migliardi and V. Sunderam. The Harness Metacomputing Framework. In *The Ninth SIAM Conference on Parallel Processing for Scientic Computing, S. Antonio (TX)*, 1999.

[21] S. Pota, K. Kuntner, and Z. Juhasz. Jini network technology and grid systems. In *MIPRO 2003, Hypermedia and Grid Systems, Opatija, Croatia*, pages 144–147, May 2003.

[22] V. Quema, R. Lachaize, and E. Cecchet. An asynchronous middleware for grid resource monitoring. *Concurrency and Computation: Practice and Experience, special issue on Middleware for Grid Computing*, 16(5), 2004.

[23] Sun Microsystems. The JNDI tutorial. `http://java.sun.com/products/jndi/tutorial/`.

[24] V. Sunderam and D. Kurzyniec. Lightweight self-organizing frameworks for metacomputing. In *Proc. of 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02), Edinburgh, Scotland*, pages 119–122, 2002.

[25] I. Taylor, M. Shields, I. Wang, and R. Philp. Distributed P2P computing within Triana: A galaxy visualization test case. In *IPDPS 2003 Conference*, April 2003.

[26] Y. Teo and X. Wang. Alice: A scalable runtime infrastructure for high performance grid computing. In *Proc. of IFIP International Conference on Network and Parallel Computing*. Springer-Verlag, 2004.

[27] T. Tyrakowski, V. Sunderam, and M. Migliardi. Distributed name service in Harness. In *The 2001 International Conference on Computational Science (ICCS), San Francisco, USA*, 2001.

[28] U. Varshney and R. Jain. Jini home networking: A Step Towards Pervasive Computing. *IEEE Computer*, pages 34–40, 2002.

[29] S. Verma, J. Gawor, G. Laszewski, and M. Parashar. A CORBA Commodity Grid Kit. In *2nd International Workshop on Grid Computing*, Denver, Colorado, 2001.

[30] Y.Huang. JISGA: A Jini-based Service-oriented Grid Architecture. *The International Journal of High Performance Computing Applications*, 17(3):317–327, 2003. ISSN 1094-3420.

Biographies

Dirk Gorissen received his M.Sc. degree in Computer Science from the University of Antwerp (UA), Belgium in 2004. In 2005 he worked as a PhD student in the Computational Modeling and Programming research group (CoMP), at UA, researching resource management within Jini-based lightweight grids. Currently he is active in the research group Computer Modeling and Simulation (COMS), also at UA, while simultaneously taking a Masters course in Artificial Intelligence at the Catholic University of Leuven. His research interests lie at the intersection of metamodeling (adaptive sampling and modeling techniques), distributed computing (Nimrod, Globus, ...) and artificial intelligence (Self Organizing Maps, evolutionary algorithms, ...).

Piotr Wendykier is a Ph.D. student in Mathematics at Emory University, USA. He received a Masters in Computer Science from Adam Mickiewicz University, Poland in 2003. After graduation, he spent one year in a commercial company where he developed Geographic Information Systems. At the same time, he was also involved in the cryptology program conducted by the Department of Discrete Mathematics at Adam Mickiewicz University. Since May 2004 he has been worked at Emory University in group lead by Professor Vaidy Sunderam. His research interests focus on a cryptology (Internet Banking) and distributed computing (H2O, HDNS).

Dawid Kurzyniec received MS degree in Computer Science from AGH University in Kraków, Poland, in 2000. He is currently a Ph. D. student in the department of Math and Computer Science at Emory University, Atlanta. His research interests include heterogeneous distributed systems, concurrent processing, and security. He is the author of over 20 conference and journal publications related to distributed metacomputing, and the main architect of the H2O metacomputing system.

Vaidy Sunderam is a professor of Computer Science at Emory University. His research interests are in wireless networks and mobile computing systems, parallel and distributed processing systems, and infrastructures for collaborative computing. His prior and current research efforts have focused on system architectures and implementations for mobile computing middleware, collaboration tools, and heterogeneous metacomputing, including the PVM system and several other frameworks such as IceT, H2O, and Harness. Professor Sunderam teaches computer science at the beginning, advanced, and graduate levels, and advises graduate theses in the area of computer systems.