Vinci: A Service-Oriented Architecture for Rapid Development of Web Applications

Rakesh Agrawal, Roberto J. Bayardo Jr., Daniel Gruhl, and Spiros Papadimitriou*
(ragrawal@acm.org, bayardo@alum.mit.edu, dgruhl@almaden.ibm.com, spapadim+@cs.cmu.edu)
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Abstract

Vinci is a local area service-oriented architecture designed for rapid development and management of robust web applications. Based on XML document exchange, Vinci is designed to complement and interoperate with wide area service-oriented architectures such as E-Speak and .NET. This paper presents the Vinci architecture, the rationale behind its design, and an evaluation of its performance. Specifically, we show how systems architected with Vinci are developed quickly, scaled effortlessly, and easily moved from prototype to production.

1. Introduction

"It's two weeks until the beta release of the site. All you have is a COBOL database, an Excel spreadsheet, a financial consulting program written in LISP, two Pentium class machines, a rack of RS6000s, a team of five developers and a 10'pole. What do you do?"--Apologies to *Zork*

Today's development cycles for web applications such as portals and marketplaces are short, and getting shorter. In some sense, these applications are never "done". The best they can hope for is "sufficient for now", with continuous improvements and enhancements as new requirements and features become apparent. This style of development contrasts strongly with more traditional models of software development involving large teams of coders who coordinate around a schedule of builds.

This new web-centric style of software development places new demands on the software development infrastructure. Because components in such systems are changing constantly, the infrastructure should allow loose coupling between them. Changes or enhancements to server components should not require modification, recompilation, or even notification of client code unless there is a significant change in the specification. In many cases, operational clients should not even have to be restarted. Such loose-coupling of distributed components reduces coordination overhead, fostering faster parallel development.

The infrastructure should support rapid prototyping as well as an easy transition from prototype to production. This transition often means moving a component to a different machine and operating system, and/or reimplementation of the component in a more efficient language. It may also mean replicating components responsible for performance bottlenecks or to improve quality of service, and employing meta-structures for load balancing across them and caching their results.

Finally, the infrastructure should be light-weight in terms of execution speed, code base, and memory footprint. We envision complex applications comprised of hundreds of computing services scattered across a LAN. For such an application to perform well, it is paramount that the interactions between the components be efficient and extensible.

In this paper, we describe and evaluate the Vinci architecture, which was created to address the requirements outlined above. Vinci bears a resemblance to some of the emerging wide-area service-oriented architectures such as E-Speak [HP01], Jini [A+99], or .NET [M00]. At its core, Vinci is based on non-validated XML document exchange in order to allow for loose connections between distributed components. It differs from these other distributed systems architectures in that it dispenses with much of the overhead associated with security, verification, and correctness checking associated with communication between untrusted parties, relegating these tasks to a gateway. As a result, Vinci allows very fast communications at the thousands of requests per second level between a broad collection of trusted local services. It can thus be viewed as a service-oriented architecture for intra-net application development that complements and inter-operates with heavier weight wide-area service-oriented systems. Vinci is language and platform neutral. This allows existing software packages to be strung together easily regardless of the language or platform required. In addition, this allows developers to select the language and platform appropriate to the task.

1.1 Related Work

A *service oriented architecture* (SOA) combines the ability to invoke remote objects and functions (called "services") with tools for dynamic service discovery, placing an emphasis on interoperability. Examples of service-oriented architectures include HP's E-Speak [HP01], Sun's Jini [A+99] and ONE [Sun01] technologies, and SOAP/UDDI [Box+00] [AIM00]. While the term "service-oriented architecture" is, to our knowledge, a somewhat recent one, related technologies have been the topic of extensive research since at least the early 1970's. One such technology is the remote procedure call (RPC) [BN84], which allows applications to invoke remote functions as if they were local functions. This feature isolates the developer from differences in operating systems, implementation languages, and network protocols used between the various hardware and software components that comprise the application.

Remote procedure call infrastructures have evolved into more feature-rich distributed object protocols such as CORBA [OMG00]

*Current affiliation: Carnegie Mellon University, Dept. of Computer Science

WWW10, May 1-5, 2001, Hong Kong.
ACM 1-58113-348-0/01/0005.

and Java's RMI [Sun99], which allow invoking object methods (functions which encapsulate state as well as behavior) in addition to traditional functions. Architectures such as Microsoft's (D)COM/COM+ [M96] and Java's Enterprise Java Beans [Sun] exploit RPCs and remote objects to support distributed component-based software engineering. There are also XML-based protocols (e.g. SOAP [Box+00] and XML-RPC [U99]) that can allow interoperation between different distributed object systems.

A common problem in building distributed applications is difficulty in evolving the application. Most of the service-oriented architectures listed above purport to reduce application rigidity by allowing the application to locate services dynamically through service registries that describe service input and output interfaces, among many other details. As of now, the main purpose of these service registries appears to be for end-user location of services that perform a desired task. In contrast to these wide-area service architectures, Vinci provides direct support of loose-coupling through environment services and conventions. Vinci also aims to provide a higher level of performance and quality of service -- one suitable for interactive web applications in addition to back-end systems integration.

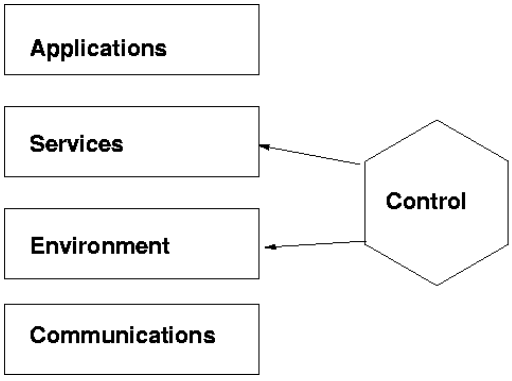


Figure 1: Vinci components.

2. Architecture

The Vinci system is composed of a number of levels (Figure 1). First, there is the communication level, which employs a protocol named *XTalk* for exchanging XML documents. On top of this is a series of conventions and environment services that simplify the task of sending messages to services and receiving results. Task specific services are built atop this environment infrastructure, and these services are monitored and restarted as needed by a control system. The top level consists of applications which are built by stringing together task-specific services. The remainder of this section describes first three levels and control layer. The following section provides an application-development case study.

2.1 Communication Layer

Vinci requires components and services to communicate by exchanging encoded XML documents. The easiest way to see how this works is with an example. Suppose a client wishes to get information on a book from the library server. It connects to the server and sends the following document.

```
<QUERY
xmlns:vinci="http://vinci.almaden.ibm.com/2000/vinci">
 <vinci:COMMAND>lookup</>
 <TITLE>Zen and the Art of Motorcycle Maintenance</>
</>
```

The server evaluates this document and returns:

```
<RESPONSE>
 <ISBN>0553277472</>
</>
```

Note that unlike SOAP with its default encoding rules [Box+00], there is no explicit typing of data in Vinci messages. There are two reasons for this approach. First, the sending of typing information in every message is an unnecessary overhead since this information is already implicit in the input and output schema definitions of the service, of which we assume the client is knowledgeable. Second, including typing information within each message complicates the evolution of both clients and servers because changes in this information must be synchronously distributed and incorporated into their implementations.

Vinci clients and services are, in contrast, expected to access their document models *declaratively* instead of immediately forcing them into some rigid typed structure. By declaratively, we mean through non-positional, associative access, e.g. an XPath expression [CD99] involving the tag names of interest. In effect, any tags present in a document that a server or client does not understand will be ignored.

Consider the following simple example illustrating the benefits of this approach. Suppose that the server in our previous example is later enhanced to return the following in response to the same query:

```
<RESPONSE>
 <ISBN>0553277472</>
 <AUTHOR>Robert M. Pirsig</>
 <PUBLISHER>Bantam Books</>
</>
```

Note that since old clients access the specific fields of interest by name, they continue to function without modification. They can later be incrementally extended to exploit the information provided by the added tags when convenient. Declarative data access facilitates *loose coupling* of the application components, allowing them to be evolved without time-consuming and error-prone synchronizations between client and server code bases.

While the exchange presented above is shown in XML notation, the communication between Vinci components is not "pure" XML, but rather a semi-parsed, pseudo-binary representation of an XML document we call *XTalk*. This representation is used because most existing XML parsers are too expensive, in terms of code size, processing time and memory footprint, for use in interactive applications. Section 4 shows how this feature allows for Vinci to perform on par with optimized RPC implementations, and an order of magnitude faster than SOAP-based services. (Concerns about SOAP performance have been expressed before [Far00].)

We provide the full specification of XTalk, along with a more thorough motivation, in the Appendix. To summarize, the main advantages of XTalk over XML are speed, size, and simplicity. Speed-wise, we have found our XTalk parsers to provide at worst a 3 times speed up over a hand-optimized, bare-bones XML parser. In practice, when compared to full-blown XML parsers such as Xerces [A00], the speedup is closer to 10 times or more. Size-wise, our XTalk parsers are approximately two orders of magnitude smaller than comparable XML parsers, and memory footprints a factor of 4 times smaller. For example, in PalmOS, our basic client, server, VinciFrame document model and XTalk conversion library has a size of only 13K.

The simplicity of the XTalk specification results in a low cost of entry into creating a library that supports the protocol, making Vinci

well-suited for integrating legacy applications and systems. While conceptually similar, pure binary document formats such as WBXML [MJ99] have a primary aim of minimizing document encoding size. This is achieved through use of symbol tables and various other mechanisms which complicate the specification and resulting parser implementations.

None of these advantages are completely without cost. Our XTalk interpreters offer no type checking and conversion, well-formedness checking, or ambiguity resolution that are features of many fully compliant XML parsers. However, our view is that these features are typically unnecessary in a deployed application. Documents are necessarily well-formed and unambiguous when constructed through programmatic document models instead of by hand. And should document validation or type-checking be a concern, Vinci allows it to be plugged in as a meta-service when and where as needed (see Section 2.3).

2.2 Basic Libraries

Vinci already includes XTalk stacks in C++, Java, Perl, Python, and PalmOS. These libraries contain classes to retrieve a document off the network and translate it into an internal document representation, and vice-versa. The default document representation used by Vinci is a light-weight and fast model we call VinciFrame, which is based on FramerD [H96] with a few additions. Other document models can be easily plugged into the infrastructure by extending them with two methods for converting the document to and from XTalk. The Java XTalk binding, for example, already supports the popular and easy-to-use JDOM [JDOM] document model as a more feature-rich (but heavier-weight) alternative.

On top of the protocol stack, the Vinci libraries provide drop in structures to make creating and deploying services easy. The service-creator simply writes a function that accepts a document and returns a document. This function is then "dropped in" to one of the stock servers, which will handle the client connection management, name-service communication, threading (if desired), and other administrative tasks.

To illustrate the process of service creation more concretely, let's look at a "hello world" example. The examples here are based on the Java Vinci API, though the other language bindings can be used similarly.

```
import vinci.transport.*;

public class HelloWorld extends VinciServableAdapter {

 // (1)
 public static final String
    SERVICE_NAME = "vinci.HelloWorld";

 // (2)
 public Transportable eval(Transportable doc) {
   VinciFrame frame = (VinciFrame) doc;
   String hello_string = "Hello"+ frame.fgetString("NAME");
   return new VinciFrame().fadd("GREETING", hello_string);
 }

 // (3)
 public static void main(String[] args) {
   VinciServer server =
     new VinciServer(SERVICE_NAME, new HelloWorld());
   server.serve();
 }
}
```

The primary tasks of the service builder are: (1) name the service, (2) provide a method which accepts the input document and returns the response document, and (3) run the service. Naming the service minimally involves selecting a character string describing what the service does. We use a naming convention similar to Java's class naming method, though Vinci does not mandate anything other than that the name be representable by a Unicode character string.

Providing a method which accepts the input document and returns the document is simplified by extending the VinciServableAdapter class. The user needs only to define a method eval() which accepts and returns a document implementing the Transportable interface. By default, the input document model is the before-mentioned VinciFrame. To have another document model provided to the eval function, the service creator simply provides an appropriate document-model factory class to the Server object in step (3).

The simplest server class offered by Vinci, the VinciServer, implements multi-threaded serving of synchronous requests. There are other server classes that can support asynchronous communications, workflows, and/or more robust connection management (supporting timeouts, connection and concurrent task limits, and so on). Using one of these alternate server classes involves simply handing the Servable to the appropriate server object in place of VinciServer in step (3).

An example of a client that invokes the above service is provided next. The document provided to the server is explicitly constructed from a document model. Optionally, Vinci provides stub code generators that accept XML-Schema specifications of the service interface and produce native language functions for invoking the service.

```
import vinci.transport.*;

public class HelloWorldClient {

 public static void main(String[] args) throws Exception {
   VinciFrame query = new VinciFrame()
     .faddString("NAME", args[0]);
   VinciFrame response = VinciClient.sendAndReceive(query,
     HelloWorldService.SERVICE_NAME);
   System.out.println(response.fgetString("GREETING");
 }
}
```

The Vinci client classes support transparent name resolution and fail-over for robustness. We discuss the details of these steps in the following subsection. For applications that require secure communication, Vinci also provides SSL versions of the servers and clients. However, the relatively high overhead of an SSL connection is somewhat at odds with the Vinci philosophy, so a better security model would be the firewall one, which we discussed in the following subsection.

2.3 Environment

Vinci provides an environment which takes care of many of the administrative details of running a distributed system, such as where to find services, monitoring of critical services (with automated emergency notifications), and service lifetime management (starting, stopping, moving, and restarting). It also provides facilities for scale-up (parallelism, spraying, load balancing, caching) and application debugging (document validation, message logging). Most of these functions can be controlled through the Web interface appearing in Figure 2.

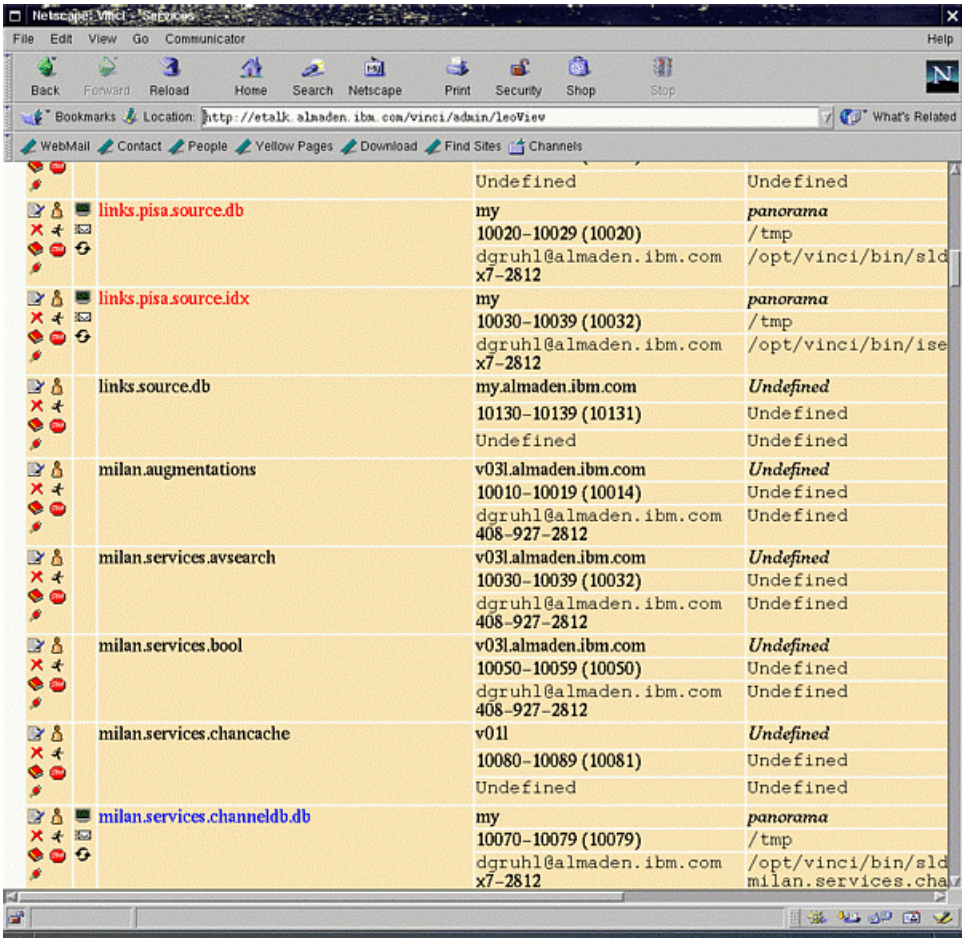


Figure 2: A web-based service browser and control application.

The first of these administrative details is what is becoming known as service discovery. Basically, a client would like to specify the requirements for a particular service and get back a host and port where that service may be found. Vinci supports this through the Vinci Name Service (VNS). Clients can ask VNS for a particular service in one of two ways: by name or by specifying an XPath expression which is applied to the meta data for a service. VNS can be thought of as an enhanced, local service cache, essentially an extension to wide area service discovery mechanisms such as UDDI [AIM00].

VNS knows where services are located because when a server first starts, it connects to VNS and negotiates a port on which to service requests. Though not required, arbitrary meta-data can also be registered with VNS by the service creator, including input and output schema specifications. On service startup, VNS tells the service which port it should use for serving requests. As part of deploying the service, the service-creator has the option of specifying a reserved port range on which the service is to be run. VNS absolves the service creator from being concerned with port conflicts. It also uses port rotation to support immediate restarts of down services.

Before issuing any requests, the Vinci client code transparently contacts VNS and receives a set of host and port pairs capable of providing the requested service. If a service is mission critical or heavily loaded and can be easily distributed, multiple instances can be deployed on different machines. For example, if a client wants to use a service vinci.services.Webster to look up some word definitions, it sends a request to VNS and receives a set of all the servers available capable of providing this service. It selects one at random and attempts to connect to it. If it cannot connect (for whatever reason) it tries with one of the other servers (providing a simple approach to high availability and load balancing). Once a connection is made, it sends a query document and waits until the response is received. The client can choose to keep the connection open and perform additional document exchange cycles over it, or it may terminate the connection and reopen it if needed again later. When maintaining an open connection to a single service location, a client will automatically spill-over to an alternate service location should the connection fail.

Another feature of VNS is the ability to assign priorities to individual service locations. These priorities specify the order in which service locations are handed out to clients. When VNS is asked to resolve a service, unless it is asked otherwise, it returns the set of highest priority locations that match the specified criteria. This allows meta services to “intercept” requests, which they may then choose to pass on to lower priority services.

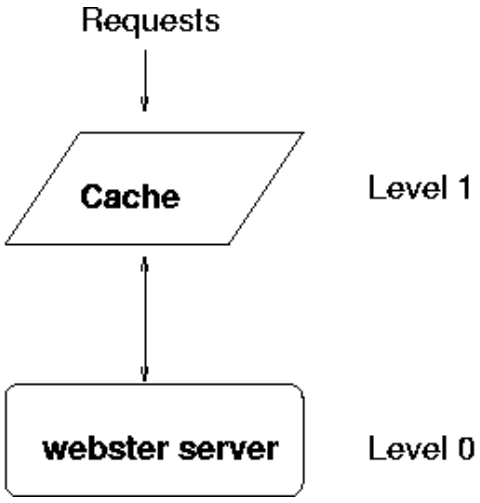


Figure 3: Caching as a meta-service.

Meta-services such as caches can be used to improve system performance without touching any code. For example, suppose a service is running at level 0, and it makes sense to cache its responses (with some cache expiration policy, for example, a day or so for the dictionary information). Instead of hacking into the service or client code itself to provide response caching, a generic caching service can be registered at level 1 (see Figure 3). Clients, which are happily unaware of the change, subsequently (and transparently) become directed to the level 1 service cache. The original level 0 server is also oblivious to the change and continues to serve requests as before, though it will now serve only those requests resulting in cache misses.

In a more extended example intended to illustrate additional benefits of configurable meta-services, a server providing validation, rerouting, etc., required by something like SOAP can be provided by chaining a series of services (Figure 4). The advantage to having separate meta-components is the ability to reuse and replace them as appropriate. For example, during development a validator might be layered on a service to make sure all the clients were sending appropriate requests. In our validation framework, each service name can be associated with an arbitrary set of constraint checkers, along with arguments (stored as documents) for each checker. A main constraint registry holds all this information. Each constraint checker is a separate service; the base environment services include a standard XML-Schema [Fal00] checker. Once the system is ready to transition to production, this meta-service can be deregistered to increase performance.

Remote Services, Proxies, and Gateways

Vinci communicates with the outside world through a proxy/gateway scheme. The proxy (Figure 5) is set up to provide an XTalk connection within the LAN, receive requests on it, dispatch heavier weight queries (such as SOAP) across the WAN, receive the response, and reformat it back into XTalk for return. The gateway (Figure 6) performs the reverse. It takes in WAN requests, acts as a client querying internal services, and returns the results over the WAN protocol.

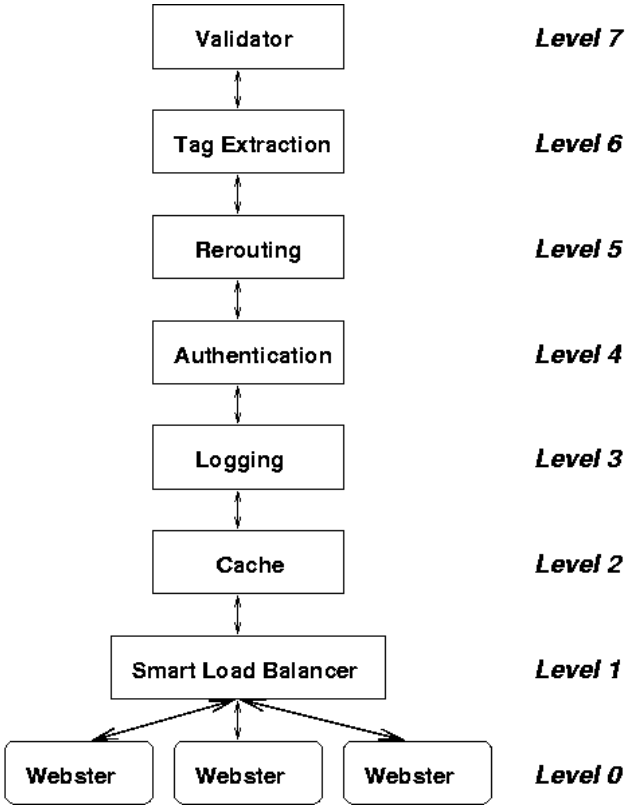


Figure 4: Enhancing a Vinci service by stacking prioritized meta-services.

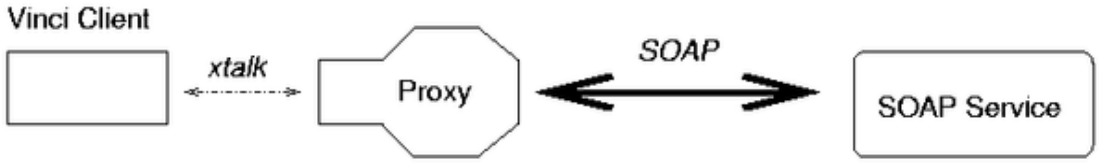


Figure 5: Vinci clients can connect to remote services offered on protocols such as SOAP using a proxy.

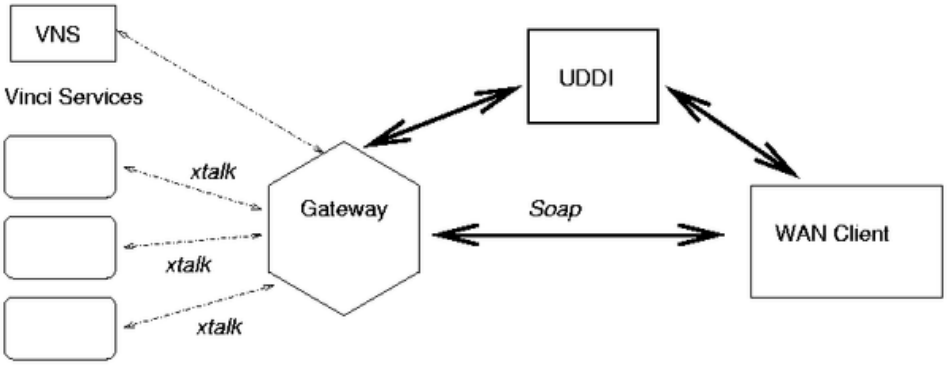


Figure 6: Vinci services can be provided outside the local area using a protocol such as SOAP and a gateway.

The gateway allows the user to specify rules as to who is allowed to request what services, how much work will be queued, limits on the types of requests, etc. We provide HTTP and e-mail gateways into our Vinci service collection. The e-mail gateway is open to the public, and currently publishes several algorithmic services to the data-mining and propositional logic research communities: http://www.almaden.ibm.com/cs/people/bayardo/vinci/index.html.

We also provide end-user interfaces to many of our Vinci services via web forms, thin GUI clients, and a peer-to-peer instant messenger application called VinciP2P (itself built from Vinci components). VinciP2P can be thought of as a "shell" into a distributed operating system of services. Automated user agents accept command-line-like command strings, convert them into the necessary document invocations, and return the response (or a link to the response if it is too large). We and others outside our project have published a number of useful services to the members of our intranet in this manner, including document conversion (ps2pdf, pdf2text, image2text, wav2mp3), IBM event information, IBM employee directory information, language translation, data-mining, and radio station monitoring (users can be notified via instant message when specified keywords are mentioned in any of multiple internet-radio broadcast streams). An interaction with the IBM Almaden event information service is depicted in Figure 7. VinciP2P implements a mini-HTTP server so that large and unstructured service inputs (e.g. a PostScript document provided to ps2pdf) can be "pulled" using HTTP at the convenience of the service instead of eagerly pushed via XTalk. (A robust, multi-threaded and optimized HTTP download service is one of the basic Vinci environment components.)

2.4 Control Layer

With any collection of multiple machines running many services each, the question of quality of service, availability, and server status are of paramount concern. Vinci employs a two-prong approach to this problem. First, each machine in the Vinci network can run a starter service. This service listens on a fixed port and can start up other services. These other services can either be started directly, or via a per-service "nanny" which forks the service and immediately restarts it if it crashes or becomes unavailable.

Second, there is a local agent which periodically pings each VNS registered service with a status query, and takes action if no response is received. This action can range from sending an email or instant message to the service owner, through trying to restart the service automatically using a starter. Precisely what action is taken is specified in the VNS meta-data for the service. In addition to taking action, the control layer also provides a web interface to allow an "at a glance" summary of the system status (Figure 2).

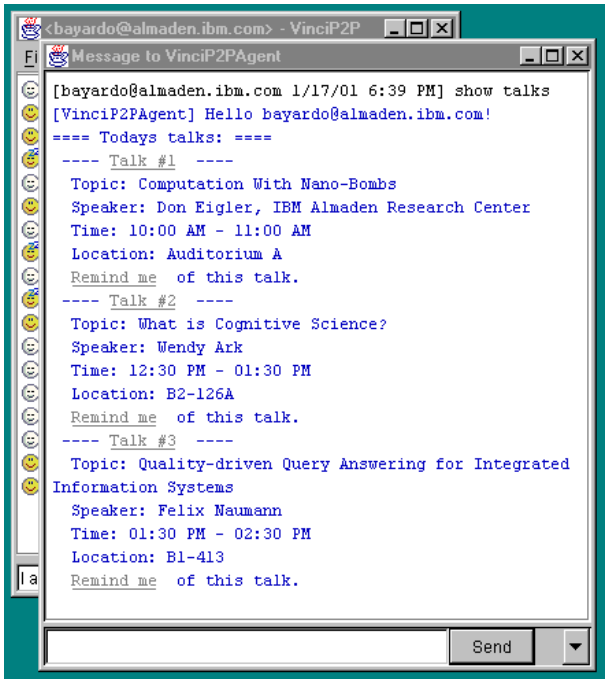


Figure 7: VinciP2P -- A peer-to-peer messenger and file-sharing tool exposing various Vinci services to end-users.

3. Application Development: A Case Study

A primary goal of the Vinci system is to make this section as short as possible. Since the "heavy lifting" for most applications is done by stock services, many content presentation applications in the Vinci domain are little more than JSPs that thread a number of them together.

For our case study, imagine you have been given the task of supplying news portlets to a company portal page. The news you will be presenting comes from a variety of sources, ranging from various web sites to contracted news feeds. Your task is to "normalize" these sources and provide a way that high level selection criteria can be applied to select the content for portlets as

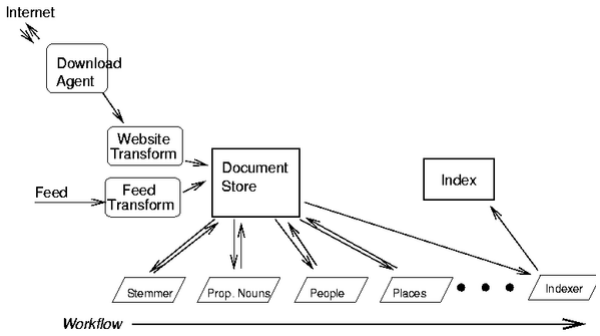


Figure 8: A Vinci content creation configuration.

varied as "IBM in the News" and "Latest Advances in Biotechnology". This task breaks into two pieces: Getting the data into a system and augmented, and searching over this augmented data and formatting it for presentation. We will consider each in turn.

Content Creation

A typical goal in content creation is to take content from some source form, and transform and augment it to make it more useful. In our example, the source data is a news article. Figure 8 depicts the transformation and augmentation process. Step by step, we have:

1. The article is acquired, either by being pushed in from a feed, or pulled in from a website. For website pulling, the HTTP download service is used to simplify issues such as timeout, threading, robots.txt file handling, and so on.
2. The article is normalized. This is done by converting it to a standard XML like representation with a necessary set of tags included (BODY, HEADLINE, SOURCE).
3. A configuration is created to take the document through a number of services. Each is called in turn to and identifies features in the body text (or in the already extracted tags; e.g. the PERSON agent looks at the tags added by the PROPER_NOUN agent). These features are written back into the source document as additional tags.
4. Once the data is fully augmented, its keys are indexed. Note: this step is not necessary if a document store is used which supports high speed querying of XML documents. We will assume for this example that this is not the case.

Content Selection and Presentation

A service for generating the "IBM in the News" portlet should return an HTML fragment for inclusion in the portal page (e.g. through a call to a URL as in JetSpeed [JET]). Such an application can be implemented as a simple JSP (or other CGI), as in Figure 9. This JSP assembles a query document that specifies the search terms, as well as which formatting service to use. In this case, an HTML formatter is specified to produce data suitable for direct-inclusion into a web page. If the target was, say, a cell phone, a WML formatter could be used instead.

The query document is sent to a boolean search service, which consults the index created earlier to identify the document identifiers of the relevant articles. These are then passed to the formatter, which retrieves the articles from the document store and formats them appropriately. Responses are then propagated back to the JSP for inclusion in the portal page.

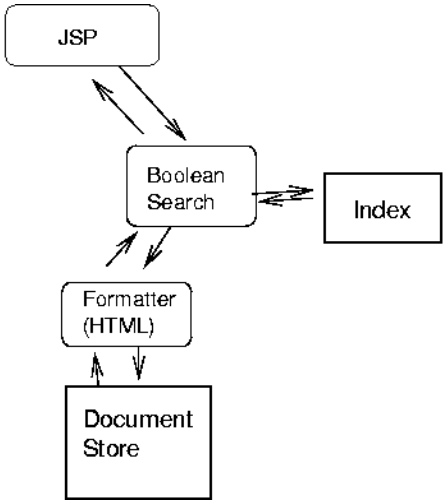


Figure 9: A JSP for content selection and presentation.

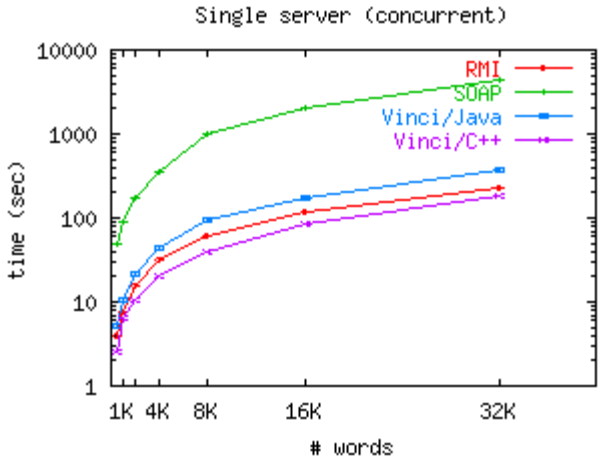
4. Performance Evaluation

The intent of this section is to quantify and contrast Vinci performance through a replicable and controlled experiment. We chose a synthetic task which is intended to be representative of a moderately CPU-intensive query that might be generated by a Web request, and whose granularity could be easily adjusted. This task involves sending a query containing a random number seed and desired result set size to a service that selects a random set of words of the specified size from /usr/dict/words, sorts them alphabetically, and returns them.

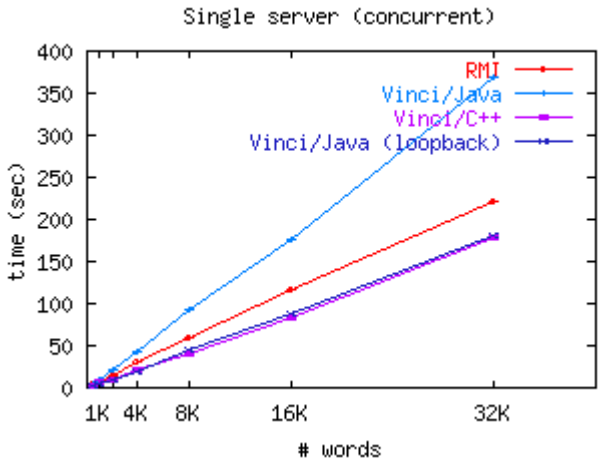
In our measurements, we have a single client perform 100 requests for a fixed result set size (500, 1000, 2000, etc., up to 32000 words), using 10 different, uniformly distributed values for the random seed. The client sends requests in batches of two concurrent requests (or four, when two servers are involved). In the Vinci experiments, the client is always our Java implementation, and the service implementation is either Java or C++ as specified. The details of our machine and network configuration are as follows:

- **Network:** 10/100Mbit switched Ethernet
- **Main server:** Intel PII 366MHz, 128Mb RAM, 100Mbit NIC
  This machine ran all services (Vinci, SOAP and RMI) for single-server measurements.
- **Main client, secondary server:** Intel PII 350MHz, 192Mb RAM, 100Mbit NIC
  Primary client for single-server measurements; also ran VNS and all validation-related services.
- **Secondary client:** Intel PII 250MHz, 64Mb RAM, 10Mbit NIC
  Client for dual-server Vinci measurements.

No other machines were present on the network. We used Sun's JDK 1.3, Xerces 1.3.0, Apache SOAP 2.0 and Jakarta Tomcat 3.2.1 running under Linux.



(a) All (log-scale)



(b) SOAP excluded (linear)

Figure 10: Single-server performance

Results

Our main set of measurements for baseline performance (see Figure 10 and Table 1) uses a single, multi-threaded server to answer all requests. As can be seen, the Java/Vinci implementation suffers only a slight slowdown (about 40%) when compared to Java RMI, which is to be expected since interpreting a Vinci document model is slightly more costly than the low-level serialized object format exchanged by RMI.

Vinci, however, offers numerous advantages and flexibility not afforded by RMI. For one, Vinci was designed with easy service composition in mind and also provides a basic set of environment services (such as caching and validation). This allows building numerous *service pipeline* configurations to enhance performance and/or functionality.

Compared to SOAP, the advantage is dramatic. SOAP message creation and parsing is a fairly involved and complicated task of generating an envelope, filling it, verifying it, extracting the necessary parameters, and so on. SOAP also typically needs to open a separate HTTP connection for each request. An order of magnitude penalty is incurred by performing these operations, most of which are unnecessary in a local-area application.

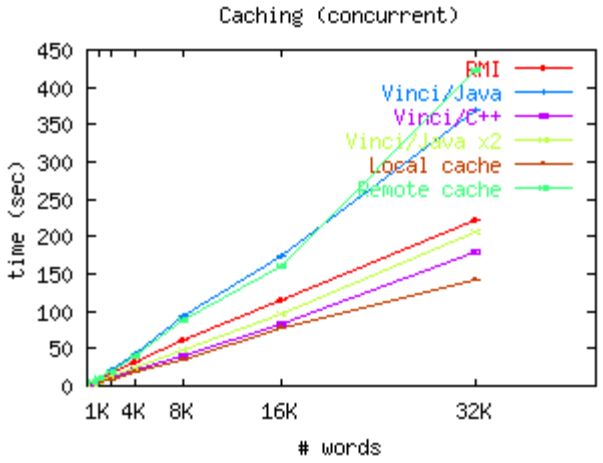
Not surprisingly, the Vinci/C++ service implementation provides performance much better than the others. This is indicative of the typical development path for Vinci: prototype in a convenient language, then rewrite in a fast one if performance is an issue.

Protocol	Time (sec)	Multiple
RMI	30.9	1.00
SOAP	349.8	11.32
Vinci/Java	43.9	1.42
Vinci/C++	20.9	0.68
Vinci/Java (dual server)	24.9	0.81
Vinci/Java (local cache)	17.6	0.57
Vinci/Java (remote cache)	40.2	1.30
Vinci/Java (validated)	268.5	8.69
Vinci/Java (validated + cached)	41.5	1.34

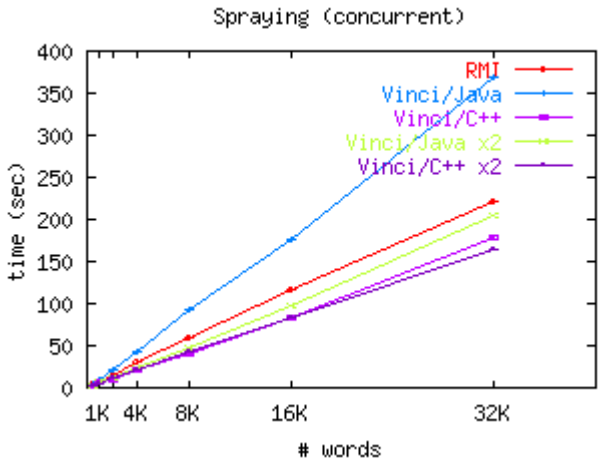
Table 1: Relative speed factors (result size: 4000 words)

We next compare a number of service pipeline configurations to the baseline performance established above. Recall that building the pipeline is a simple task of starting the necessary environment services. One such environment service, result caching, can improve performance by avoiding re-execution of expensive operations. The generic Vinci caching service, which is written in Java, can be inserted anywhere along the service pipeline. It can be placed closer to the server to increase the chance that multiple clients reuse the same result, or closer to the clients to decrease network traffic and communication penalties. We evaluate using both a local (i.e. client-side) and remote (i.e. server-side) cache. In our configuration, the hit ratio is 90%. The times are shown in Figure 11a. In this case a local cache provides performance superior to the uncached Vinci/C++ implementation. Since the network is the bottleneck in this case, a remote cache is not as effective but still improves performance (except for the largest granularities where it appears the additional Java memory management costs outweighed any benefits of caching). A C++ implementation of the cache service would likely improve performance.

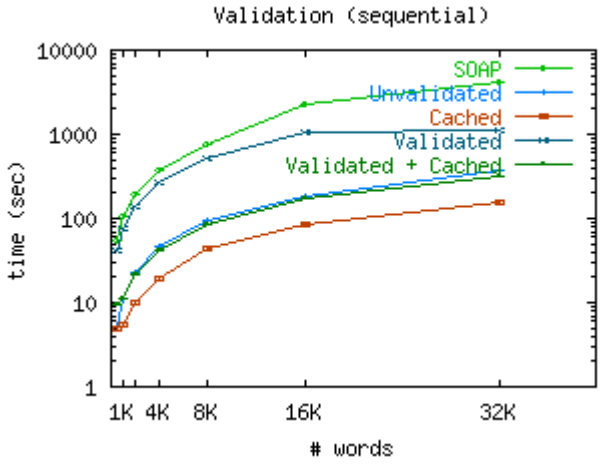
Vinci supports load-balancing in numerous ways, each requiring only modest effort. The simplest way is to register multiple service instances in VNS and distribute client connections among these. Because a Vinci client automatically distributes connections randomly among all equivalent service locations, the only step required for this is starting the new instances. Another option is to use a *spraying service* that implements sophisticated load-balancing algorithms. Such a service could be added as necessary in the pipeline. In our experiments, we have the client explicitly connect to the two available service locations with a round robin policy. This technique is sufficient for a number of situations with relatively heterogeneous request sizes. With the Vinci/Java version of our service, adding an additional service location doubles the performance. Adding another C++ server did not improve performance significantly. We suspect this result is from the bottleneck moving from the server to the client side.



(a) Caching



(b) Spraying (load-balancing)



(c) Input validation (log-scale)

Figure 11: Performance of various Vinci service pipeline configurations

We lastly show the impact of adding input validation. The validator's task is to intercept requests to a particular service and contact the main validator registry to check all constraints for that service. The flexibility offered by our validation framework comes at a price, since checking all constraints involves a number of XTalk requests (one to each checker). However, we believe the tradeoff is justified; validation is typically used infrequently, and primarily on services exported to untrusted clients via a gateway. Figure 11c shows performance with input validation. There is a significant penalty involved, but Vinci with validation as a meta-service is still significantly faster than SOAP. Also, note that inserting a cache in the pipeline almost eliminates the penalty.

5. Conclusions

Vinci supports fast development of efficient, scalable, and evolvable applications composed of loosely-coupled services that communicate via XML document exchange. The cost of entry into creating a service that can fully participate in the Vinci infrastructure is kept low by delegating the responsibility of authentication, security, XML parsing and translation to heavy-weight gateway nodes. Services perform well by exchanging an easily interpreted, semi-parsed XML document representation called XTalk. Vinci service invocation overhead is roughly equivalent to that of a light-weight RPC protocol, and over an order of magnitude less than XML-based protocols such as SOAP/HTTP. This allows Vinci services to participate in interactive applications, such as Web portal front-ends, in addition to back-end integration tasks. Extensive environment services and control functions allow components to be distributed, cached, validated, and logged without affecting servers or clients, or even requiring they be restarted. Developers can thereby evolve running applications without even taking them out of service.

The rapid adoption of world-wide-web protocols was arguably brought about by the simplicity of HTML and the resulting low barrier to creating a functional web page. Vinci aims to bring this property to service creation and communication by allowing complexity to be layered on cleanly, as needed, with minimal impact on the code base.

Appendix: XTalk

Motivation

Rumor has it that XML was intended to be a variant of SGML simplified to the point where any DPH ("Desperate Perl Hacker") could write a parser for it over one weekend. While XML is undeniably far simpler than SGML, the reality is that it remains of sufficient complexity to make parser implementation difficult -- so much so that large open source efforts are dedicated to XML parser implementation [A00]. Another side effect of this complexity is that parsing XML requires significant computational overhead, at least compared to the overhead of simple services which may wish to communicate by exchanging XML documents.

XTalk is a pseudo-binary XML format intended to make the XML parsing task even more simple than what was originally envisioned by the XML creators. It is not, however, intended to be a replacement for general XML documents. Indeed, we expect textual XML to be the mainstay of document exchange. XTalk is best used as an intermediate XML representation exchanged by high-performance, distributed services that run on anything and everything from the hand-held to the mainframe. The representation may also be suitable for storage in persistent XML stores.

We realize that any proposal for a non-textual document representation may be met with considerable resistance, as it deviates from the primary human readability criteria that has motivated specifications such as SGML, HTML and XML from the start. Nevertheless, the need for standardizing on a non-textual representation has been expressed numerous times on W3C discussion lists and elsewhere, including one in which Tim-Berners Lee has expressed support for the idea [B-L99]. XTalk attempts to deviate from the human readability criteria as little as possible by representing only structural aspects of the document in binary, and leaving all data components in the standard UTF-8 character format.

In order to ensure that XTalk is capable of representing all semantic components of an XML document, the XTalk specification was developed as a serialization format of the XPath XML data-model [CD99]. The design goals of XTalk are both simplicity and efficiency. Materializing an XTalk representation into a document model such as DOM or the XPath data model requires under 100 lines of code, and even fewer lines are required for converting JDOM, DOM [DOM] or XPath data models into XTalk. Converting XTalk into XML is also a (nearly) trivial task. We believe this strongly mitigates any concerns of non-human readability.

Much of XTalk is also motivated by the Canonical XML specification [Boy00], even though the goals of these specifications are orthogonal. Canonical XML is a textual XML representation that allows one to determine if two (or more) physically different XML documents are logically equivalent by simply checking if their canonical representations are identical. Canonical XML documents use only a subset of XML constructs, thereby eliminating many of the complexities fully XML-compliant parsers must deal with. Just to name a few, this includes:

- DTD's (and associated complexities including default attributes and entity refs.)
- multiple character encodings
- multiple ways of encoding character data (e.g. <!CDATA[&... vs. & vs. &)
- different quoting characters
- different ways of defining empty elements (single tag vs. start/end tags)
- comments
- whitespace within tags

Indeed, canonical XML itself seems to be a reasonable candidate for the XTalk specification, since writing a parser for the subset of XML that may appear in a canonical representation is much simpler than writing fully compliant XML parser. However, we believe that canonical XML remains sufficiently difficult and computationally expensive to parse that further simplifications are warranted. We have designed XTalk so that converting canonical XML document to XTalk results in no information loss; the canonical XML document can always be materialized from its XTalk representation without change.

XTalk differs from canonical XML primarily in that it explicitly encodes the node structure of the document in binary, rather than requiring the parser to extract structure information from textual markers. This allows the representation to encode strings without the need for character references. While additional (time/space) optimizations of the encoding are possible, we believe most would result in excessively complicating parser implementation. The intent of XTalk is to strike a good balance between parser simplicity and representational efficiency. It does this by leaving all the data components of the XML document in character (UTF-8) format, and representing only the document's tree structure in binary. While this *pseudo-binary* representation does not yield as significant a reduction in space as a pure binary representation such as WBXML [MJ99], we believe standard compression techniques can be used in concert with XTalk (or XML) to better achieve size-reduction goals without significant added complexity or performance penalty. A W3C comment on the WBXML specification provides a similar argument [C99].

The specification of XTalk appears below in BNF. The doc non-terminal encodes the root node of the XPath spec, the element non-terminal the element node, the pi non-terminal the processing-instruction node, and the attr non-terminal encodes both attribute and namespace nodes. There is no separate non-terminal for the namespace node because XTalk represents in binary only the structure of the XML document. Since XML namespace declarations are structurally equivalent to attributes, one non-terminal is sufficient.

Note that every leaf component of the structure is preceded by the number of bytes occupied by the component, and in every element component, the number of sub-components precedes the list of sub-components. This simplifies the parser's memory allocation and streaming strategies since at any point the parser knows how many bytes to read and allocate in order to consume the next component. There is no need to ever check for end of string, end of stream, or any other terminal indicators.

Specification

doc ::= 'X' versionid int ('p' pi)*
'E' element ('p' pi)*

versionid ::= byte

element ::= string int attr* int child*

child ::= ('s' string)
| ('E' element)
| ('p' pi)

attr ::= string string

pi ::= string string

string ::= int utf8

utf8 ::= (byte array of valid utf8 character data)
int ::= (4 byte big-endian unsigned integer)

Constraints:

doc:
Total number of pi occurrences must equal the value of the int + 1.

element:
String must abide by the XML 1.0 restrictions on tag names.
Total number of attr occurrences must equal the value of the preceding int.
Total number of child occurrences must equal the value of the preceding int.

attr:
First string must abide by XML1.0 restrictions on attribute names.
Second string must abide by XML1.0 restrictions on normalized attribute values.

References

[A00] Apache Project, Xerces Java Parser. http://xml.apache.org/xerces-j/,2000.

[AIM00] Ariba Corp., IBM Corp., and Microsoft Corp.. *UDDI Technical White Paper*, http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf, Sept. 6, 2000.

[A+99] Arnold, K.; O'Sullivan, R.; Scheifler, W.; Wollrath, A.. *The Jini Specification*. Addison-Wesley, Reading, Mass. 1999.

[B-L99] Berners-Lee, T. W3C discussion list posting. http://www.lists.ic.ac.uk/hypermail-archive/xml-dev/xml-dev-Sep-1999/0839.html, 17 Sept. 1999.

[BN84] Birrell, A.D. and Nelson, B. J.. *Implementing Remote Procedure Calls*. ACM Transactions on Computer Systems 2, 1(Feb. 1984):39-59.

[Box+00] Box, D.; Ehnebuske, D.; Kakivaya, G.; Layman, A.; Mendelsohn, N.; Nielsen, H. F.; Thatte, S.; and Winder, D.. *Simple Object Access Protocol*. http://www.w3.org/TR/SOAP/, May 2000.

[Boy00] Boyer, J. (ed). *Canonical XML Version 1.0 specification*. http://www.w3.org/TR/xml-c14n, Oct. 2000.

[Br+00] Bray, T.; Paoli, J.; Sperberg-McQueen, C. M.; and Maler, E. (eds.). *Extensible Markup Language (XML) 1.0* (Second Edition), http://www.w3.org/TR/REC-xml, Oct. 2000.

[CD99] Clark, J. and DeRose, S.. *XML Path Language (XPATH) Version 1.0*, http://www.w3.org/TR/xpath, Nov. 1999.

[C99] Conolly, D. Comment on WBXML Submission. http://www.w3.org/TR/DOM-Level-2-Core/, 1999.

[D+00] Davis, M.; Le Hors, A.; Le Hegaret, P.; Robie, J.; Wood, L.. Document Object Model (DOM) *Level 2 Core Specification*, http://www.w3.org/TR/DOM-Level-2-Core/, Sept. 2000.

[Fal00] Fallside, D. C. (ed.), XML Schema Part 0: Primer, http://www.w3.org/TR/xmlschema-0/, Oct. 2000.

[Far00] Farley, J. *Microsoft .NET vs. J2EE: How Do They Stack Up?* http://java.oreilly.com/news/farley_0800.html, 2000.

[H96] Haase, K. B. *FramerD: Representing knowledge in the large.* MIT Media Lab Technical report, 1996.

[HP01] Hewlett-Packard Corp. *E-speak*, http://www.e-speak.net.

[JET] JetSpeed, http://java.apache.org/jetspeed/site/index.html.

[JDOM] The JDOM Project, http://www.jdom.org/.

[MJ99] Martin, B. and Jano, B. (eds.). *WAP Binary XML Content Format*, http://www.w3.org/TR/wbxml/, W3C note, June 1999.

[M96] Microsoft Corp.. *DCOM Technical Overview* (http://msdn.microsoft.com/library/backgrnd/html/msdn_dcomtec.htm), 1996.

[M00] Microsoft Corp. *The Programmable Web: Web Services Provides Building Blocks for the Microsoft .NET Framework.* http://msdn.microsoft.com/msdnmag/issues/0900/WebPlatform/WebPlatform.asp, 2000.

[OMG00] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, Revision 2.4, October 2000. ftp://ftp.omg.org/pub/docs/formal/00-10-01.pdf.

[Sun] Sun Microsystems, Inc.. Enterprise JavaBeans Technology (http://java.sun.com/products/ejb).

[Sun99] Sun Microsystems, Inc. *Java Remote Method Invocation - Distributed Computing for Java.* http://java.sun.com/marketing/collateral/javarmi.html, Nov. 17, 1999.

[Sun01] Sun Microsystems, Inc.. *Sun Open Net Environment (Sun ONE) Software Architecture*, http://www.sun.com/software/sunone/wp-arch/wp-arch.pdf, 2001.

[U99] UserLand software. *XML-RPC Specification*, http://www.xmlrpc.com/spec, Jan 1, 1999.

[W] Waldo, J.. *The End of Protocols*, http://developer.java.sun.com/developer/technicalArticles/jini/protocols.html.