Distributed Indexing of Autonomous Internet Services

Peter B. Danzig, Shih-Hao Li and Katia Obraczka
University of Southern California

ABSTRACT: This paper describes the architecture and the design decisions behind a resource discovery tool that we prototyped to knit together the Internet's resource discovery fabric. We call the architecture distributed indexing or *Indie* for short. Indie consists of a directory of services and an unlimited number of *broker* databases that index their own data, data stored in other brokers, and data available from other resource discovery services. The indexing mechanism doubles as a lazily consistent data replication mechanism that can replicate the directory of services or any other broker at will.

An Indie broker automatically clusters references to related objects stored in other autonomous discovery and database services. Since Indie brokers cluster related information skimmed from thousands of scattered services, efficient exhaustive search is possible. This centralization led to the success of the *archie* file location service. In a way, Indie is a generalized archie that locates autonomously maintained data stored in different discovery services.

We believe that other discovery tools can benefit from the architectural principles that Indie illustrates and the

ability to cluster related information that Indie provides.

1. Introduction

Roughly speaking, resource discovery tools either *organize* or *search* information distributed across many repositories. Tools like the Internet gopher [1], the World-Wide-Web [3], and Prospero [11] organize information into a distributed hypertext. With these tools, people create links between relevant information which may reside on different servers. Tools like archie [7], Nomenclator [14], and netfind [15] build indices from information scavenged from various repositories and sources. This paper describes a discovery architecture that, similar to this second set of tools, automatically clusters pointers to related information obtained from other repositories and discovery services so that it can be efficiently searched. We call our prototype of this architecture *distributed indexing* or *Indie* for short [5].

1.1 Indie Architecture

Indie consists of a replicated *directory of services* and a collection of servers or *Indie brokers* that index information available from other brokers, repositories, and discovery services. Indie specializes in efficiently locating information with minimal need to organize it manually.

A *generator object* describes each Indie broker. The generator consists of a textual abstract, a boolean expression over an extensible set of bibliographic fields which we call the *generator rule,* and certain technical data about the broker. The appellation *generator rule* sums up the idea that a broker's contents is generated by executing the boolean query on lots of other discovery services and storing the results. Essentially, the rule generates a broker's database. Figure 1 illustrates a generator object.

attribute name	*attribute value*
`generator rule`	(keywords = network*)
`abstract`	Everything you wanted to know about networking but were afraid to ask.
`LoC_ranges`	TK5100 - TK6000 | QA76.6 - QA76.99
`type`	Indie broker | gateway
`location`	34 03 08 N 118 14 34 W
`object_id`	{ host:berkeley.edu port:32001 oid:1 }
`object_type`	generator
`size`	14,505, entries
`replicates`	{ host:caldera.usc.edu port:32001 oid:1 }
`directory replica`	caldera.usc.edu:32002
`created`	11:59:01 PM June 1, 1992
`status`	creating | valid | deleting

Figure 1: Generator object for a replica.

All brokers register their generator objects with the directory of services. A broker can register itself with any replica of the directory of services, but the particular replica with which a broker registers must agree to perform certain services. Initially, it must return a list of other generator objects pertinent to the new broker, but it must also agree to report changes to this list. A broker stores its list of generator objects in its *registration table* and refers to this table when choosing the set of services it will index.

A broker's administrator, via the administrator's tool, chooses which services listed in its registration table to index. The directory of services sends updates to the registration table asynchronously, so from time to time the administrator must elect whether or not to index new services. The administrator tool feels similar to network news reader programs where instead of subscribing or unsubscribing to news groups, you register or unregister with other brokers and gateways to other discovery services.

When some broker **A** decides to index some broker **B**, the indexed broker stores the generator object in **B's** *trigger table* (see Figure 3). The name trigger table should evoke the sense of an active database [17] in which specific rules are triggered and evaluated when the database changes in particular ways. When one broker registers its generator object with another broker, the indexed broker executes the generator rule and reliably forwards the retrieved set of *object descriptors* to the indexing broker. Afterwards, adding or deleting objects from the indexed broker's database may trigger the evaluation of generator rules in its trigger table, and some of these rules may forward these changes to their corresponding brokers.

The directory of services is just a specialized broker. When a broker registers itself with a replica of the directory of services, the broker's generator object is stored in that replica's trigger table. Only the updates to the directory of services that trigger this rule are forwarded to the broker.

Lazily consistent server replication is a side effect of Indie's indexing mechanism. To replicate a server, we create an Indie broker to serve as the replica, assign the replica the same generator rule as the server to be replicated, and have the replica index the server or some number of replicas of it. Since the replica shares the same generator rule as the primary copy, it fills with the same data. The replica need not directly index the primary copy, but instead can index another replica. Section 2 discusses replication and consistency issues in the context of replicating the directory of services.

Non-Indie servers attach to Indie through a *gateway* broker. Indie provides a *gateway library* consisting of a set of routines which non-Indie servers can call to communicate with their gateway broker. The gateway broker itself is just a normal Indie broker modified as necessary to communicate with the non-Indie server. It manages a trigger table, registration table, the interface to the directory of services, and the data consistency mechanism. A non-Indie server can cooperate directly and efficiently with Indie by making appropriate calls to the gateway library. If a non-Indie server does not cooperate directly, its gateway broker must communicate with it in its native protocol and poll it for updates by brute force. Section 4 describes the gateway library and discusses how to index other discovery services.

1.1.1 Atomicity, Consistency, and Recovery

Indie addresses database consistency and recovery with a timestamped augmented flooding algorithm which works as follows. An object, regardless of where it resides, is labeled with the object identifier or *oid* of the database which created it, and all trigger table entries, registration table entries, and objects stored in a broker's database are timestamped. Objects are stamped with the time at which they were first added to the broker's database. Trigger and registration table entry timestamps are updated by the consistency algorithm, described here.

Figure 2 illustrates an Indie object descriptor. An object descriptor can contain an arbitrary number of attribute-value pairs. In the figure, the last few attribute-value pairs are used by the broker's consistency algorithm. Notice that the object descriptors need not include the object itself. This lets a service advertise an object but retain control of access to it.

When an object is added to or deleted from a broker, the change causes the broker to evaluate certain rules stored in its trigger table. For each rule so triggered, the broker tries to establish a communication connection with its *peer*. Brokers are peers if either or both brokers index the other. Once communication is established, the broker evaluates the generator rule against updates with timestamps younger than the trigger table entry, forwards changes to the remote peer, and finally advances the trigger table timestamp to the timestamp of the latest update. It then transmits this timestamp to its peer, which records it in the appropriate registration table entry. If it cannot establish communication with its peer, it marks the trigger table entry as out of date. Timestamps of rules neither triggered nor marked as out of date are advanced to the current time. Finally, peers occasionally poll one another in an attempt to maintain consistency.

We describe the algorithm by example. Figure 3 shows three brokers named A, B, and C. A indexes B and C and B indexes C. The second field of the registration table indicates whether the broker is subscribed (S) or unsubscribed (U) to its peer. The second field of the trigger table indicates whether the rule is out of date (O) or not (Z).

attribute name	*attribute value*
`author`	Peter Danzig
`title`	Distributed Indexing of ...
`month`	September
`year`	1992
`abstract`	Internet resource discovery tool...
`keywords`	Indie
`LoC_number`	QA76.9.D3
`journal`	Submitted to Computing Systems
`volume`	5
`number`	3
`pages`	175-188
`note`	to be published
`pts_to`	ftp warthog.usc.edu pub/jcs.ps
`object_id`	$\left\{\begin{array}{l}\text{host:caldera.usc.edu}\\ \text{port:32004}\\ \text{oid:532}\\ \text{version:1}\end{array}\right\}$
`object_type`	object descriptor
`consistency`	hard | soft
`timestamp`	3:45:01 AM June 15, 1992
`state`	present | locked | deleting
`who_we_told`	$\left\{\begin{array}{l}\text{cs.wisconsin.edu:32004}\\ \text{expresso.cc.mcgill.ca:32004}\end{array}\right\}$
`who_told_us`	$\left\{\begin{array}{l}\text{object\_resides\_locally}\\ \text{cs.berkeley.edu:32004}\\ \text{expresso.cc.mcgill.ca:32004}\end{array}\right\}$

Figure 2: An Indie object descriptor.

The timestamp mechanism permits convenient recovery from network partition, operating system crashes, and media failure of the broker's database. It also facilitates creating new replicas. If media failure destroys the broker's database but not its registration and trigger tables, then recovery simply involves setting the timestamps on all entries of these tables to zero. The normal polling process will recover the broker's database.

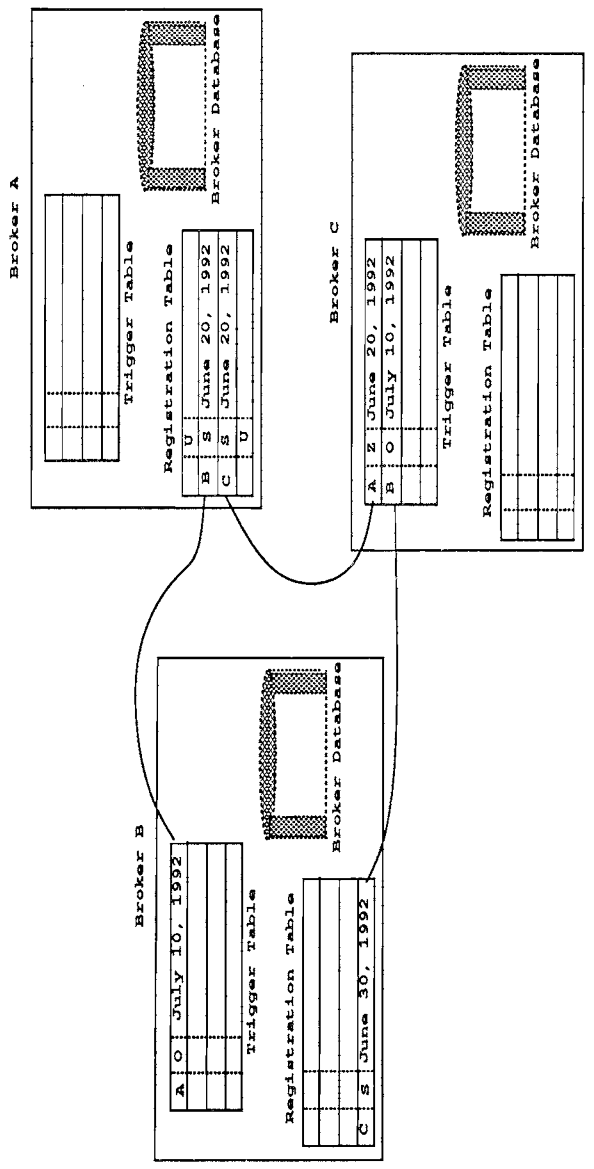

Figure 3: Timestamps between brokers.

If media failure destroys everything, then partial broker recovery starts when the broker retrieves its generator object from the directory of services and requests that the timestamp of the directory of service's trigger table entry be set to zero. This forces the directory replica to reinform the broker of services relevant to its generator rule. The broker's peers, as they poll the broker, will find that the peer relationship has been broken and will toggle the registration table entry into the *recommended* state. The broker administrator can use this as a hint to re-register with its peer. As the broker re-registers with its peers, it will refill with data. Brokers indexing the recovering broker, during polling, discover that the trigger table entry is missing, and are forced to re-register with the recovering broker. While a recovering broker may send objects to its peers that they have already received, these objects get discarded because of their duplicate oids.

1.1.2 Indexing Topology

Indie's rules can only delete and add objects to a broker's database. Thinking of Indie brokers as nodes and subscribed generators as edges, how do concerns for stability and object consistency limit the topology of the resulting graph?

In databases with triggered rules, a pernicious set of rules can trigger each other in an endless cycle. As an example, consider two databases that register the following rule with one another: "Upon updating your database, send me a new object that records the time of the update." Once either database is updated, the two rules trigger each other endlessly. Indie generator rules cannot do this because they do not create new objects, but only cause existing objects to be replicated on or deleted from the peer broker. A broker, other than increasing the object's reference count, ignores attempts to add an object already in its database; it also ignores attempts to delete an object that is not in the database. Ignored updates cannot trigger additional rules. Hence, concerns for stability do not limit the indexing topology.

As Figure 2 illustrates, Indie uses field `who_told_us`, in addition to the reference count just mentioned. This field lists all the brokers that forwarded a particular object descriptor to a broker. Field `who_we_told` is not actually stored; rather, when needed, this field is regenerated by applying the object descriptor against the generators in the trigger table.

Indie's sense of strong consistency requires that brokers delete object descriptors that are not reachable through the current indexing topology. A weaker notion of consistency based on deleting an object when a cached `time_to_live` value expires is not currently implemented. Strong consistency requires that when a broker unregisters its generator object with its peer, that it removes the peer's `who_told_us` entry from all object descriptors which the peer provided, and delete those objects whose reference count reaches zero. When a peer instructs a broker to delete an object, the broker removes the appropriate `who_told_us` entry and decrements the object's reference count. When an object's reference count reaches zero, the broker marks the item as deleted and tries to forward the deletion to peers in its `who_we_told` list (which it recreates by evaluating the rules).

Figure 4 shows the reference count of an object provided by broker **A**. On the left hand diagram, broker **B** hears about object **OID** from both **A** and **F**. So its reference count is two. On the right hand diagram, **B** still retains a copy of **OID** even though **A** deleted it because **B** thinks that **F** can still provide it. If, on account of some request, **F** tries to fetch object **OID**, **A** will inform **F** that the object no longer exists.

Indie partially solves this consistency problem with a *strong* deletion operation. Strong deletion ignores reference counts and forces deletion. When a broker deletes one of its own objects or when a broker discovers a consistency problem, it invokes strong deletion. However, this only partially solves the consistency problem. If broker **A** had actually gotten **OID** from some other broker and broker **B** unsubscribed from broker **A**, it would not be able to delete **OID** because it would still believe that **OID** was reachable from broker **F**. However, when an Indie broker attempts to fetch an object, and the originating broker says that the object no longer exists, the Indie broker invokes strong deletion.

Deletion consistency is easily achieved if the indexing topology is restricted to a directed acyclic graph. Indie can enforce a DAG topology by restricting indexing to be hierarchical, but we believe this is a heavy price to pay for data consistency.

1.1.3 Update Rate Restriction

Indie restricts the rate at which a broker can forward updates to its peers to reduce performance degradation. For example, if a broker

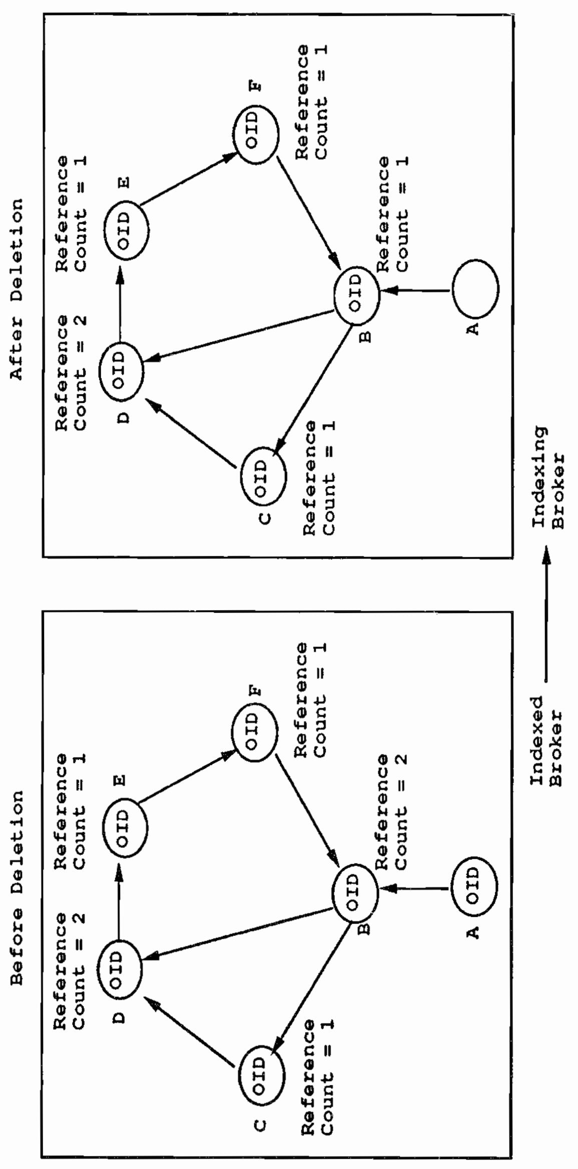

Figure 4: Topology, reference counts, and deletion consistency.

changes its generator object, it does not forward updates to its peers for several days. This restriction attempts to short circuit transients caused by broker tuning and experimentation. Although an Indie broker can change its generator object as frequently as it wants, the directory of services limits the rate at which it will register the new generator. For stability, objects added to an Indie database must meet a particular *residency time* before they can trigger the evaluation of generator rules.

1.2 Outline

The remainder of this paper describes the interface to Indie's directory of services, contrasts Indie with other discovery services and explains how Indie can index them, and reviews our Indie prototype.

2. Directory of Services

This section describes Indie's directory of services, from issues of replication, consistency, and recovery, to the programming library primitives that interface to it.

2.1 Replication and Consistency

All replicas of the directory of services are created equal, and "none are more equal than others." This means there is no primary copy of the directory of services. Instead, clients register or unregister themselves with the replica of their choice. Indie's update and recovery algorithm guarantees that all replicas eventually learn of the update.

Viewing the replicas of the directory of services as nodes and registered rules as directed edges, this graph must be connected and every edge in one direction must be paired with an edge in the other direction. Figure 5 illustrates this topology. Any update satisfies the trigger registered between replicas. All said, this implements a simple flooding based replication algorithm.

Consistency demands that the graph be connected and care be taken when changing a replica's peers. Consider what happens when a replica drops out of the graph, and then reattaches itself somewhere

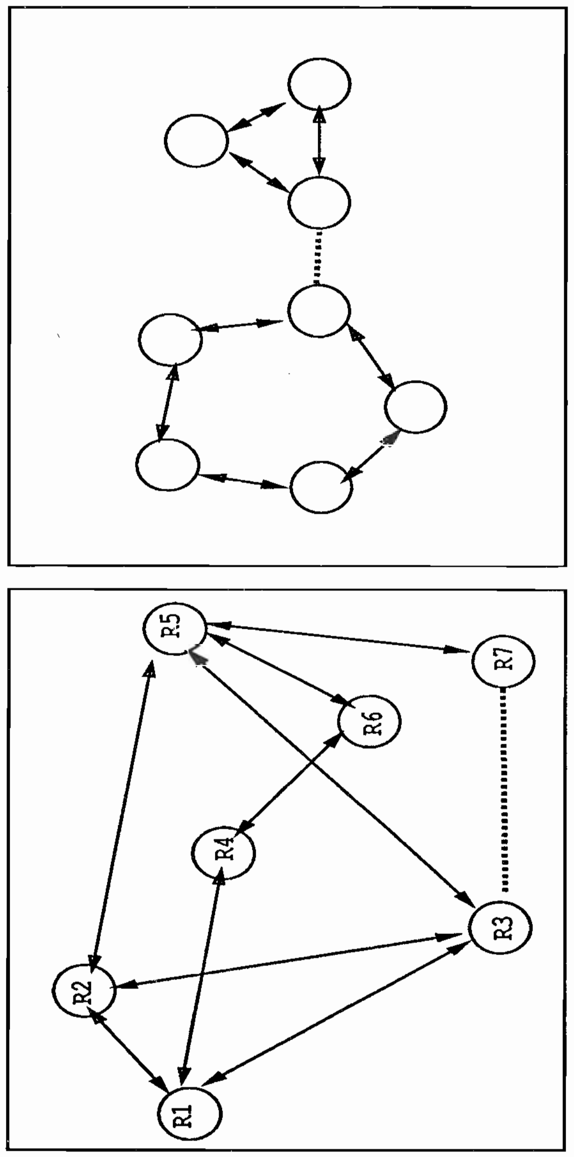

Figure 5: Topology of replicas of the directory of services.

else. Suppose that replica **R7** disconnects itself from **R5** and reconnects with **R3**. If some object is deleted while **R7** is disconnected, **R7** may reinject and reincarnate this object when it is reconnected.

Notice that reincarnation does not occur if the topology remains fixed, even if a replica is unreachable for a long time. The problem occurs when a replica that has disconnected itself from its peers reconnects and exchanges data with a new peer. The right hand side of Figure 5 illustrates this problem. If topology changes disconnect and later reconnect the graph, the two partitions exchange updates, but can also reincarnate objects deleted from the other partition.

Indie brokers (and directory of service replicas) avoid inconsistency through reincarnation by keeping a cache of deleted objects. If a peer broker tries to reinject an object in its peer's deletion cache, its peer responds by injecting a strong deletion. If a peer tries to insert an update with timestamp beyond the oldest timestamp in the deletion cache, the peer first verifies that the object exists by trying to retrieve it from the object's home broker. If the object is invalid, the peer injects a strong deletion.

The directory of services, like all Indie brokers, can add and delete objects. To simplify maintaining consistency between brokers, if you modify an object stored in an Indie database, Indie assigns it a new object identifier and deletes the old one. If desired, Indie can compose the new object identifier by incrementing the version number of the old object identifier. However, Indie treats different versions as different objects.

2.2 Interface

Indie brokers and gateways communicate with replicas of the directory of services with the following (principle) primitives:

Register (replica, generator) An Indie broker registers its *generator* with this particular *replica* of the directory of services. Neighbor replicas of the directory of services invoke this primitive symmetrically so either replica can update the other. Because **register( )** is idempotent, a broker can invoke it at any time.

Unregister (replica, generator) Removes a broker's *generator* from the directory of services. Neighbor replicas use **unregister( )** to change the flooding topology of the directory of services.

Generator [ ] =Lookup (query) Returns a list of *generator* objects that match the *query*.

2.3 Dealing with Replicas

The directory of services has a distinguished object identifier `(host: 0 port: 0 oid: 0)`, which generator objects of all replicas of the directory of services record in field `replicates`. To help find physically close replicas, generator objects can contain the service's latitude and longitude in field `location` and the boolean generator rule permits distance queries such as `(Distance < 500 miles)`. Client programs can use this field, if present, to select a geographically close replica.

3. Contrast with Other Tools

This section briefly overviews resource discovery tools currently available on the Internet, contrasts them with Indie, and discusses how Indie can cooperate with them.

3.1 Archie

Archie servers [7] index names of files available from hundreds of FTP archive sites. Users can query an archie server for file names that match specified patterns, a complete list of the FTP archive sites, or a list of the files available from specific sites.

Archie is a great success partially from its simplicity and use of available mechanisms, and because it centralizes searchable information. According to its developers, *archie* needs a more efficient consistency maintenance protocol so that topics can be distributed across different servers. This will make archie more scalable and resilient. Archie's developers also plan to build more elaborate indices and enable attribute-based searches.

3.2 The Wide Area Information Server

WAIS [9] is a full text information retrieval architecture whose clients and servers communicate through an extension of the Z39-50 standard [10]. One server is distinguished as the directory of services, but it is manually maintained.

WAIS clients can query the WAIS directory for relevant servers, and query a subset of these for pertinent documents. WAIS servers keep complete inverted indices of the documents they store, and execute full-text searches on them. In response to a query, a WAIS server returns a list of relevant document descriptors which the client displays. WAIS clients support relevance feedback to help users refine their queries. The client retrieves documents from the appropriate servers and displays them for the user.

3.3 Gopher

The Internet gopher [1] both organizes and searches distributed information. Gopher organizes information hierarchically[1], where intermediate nodes are directories or indices, and leaf nodes are documents. The root of Gopher's hierarchy is stored on host *rawBits.micro.-umn.edu* at the University of Minnesota. This is the default directory retrieved by the gopher client when first invoked.

Gopher objects are identified by their type, user-visible name, server's host name and port number, and the object's absolute path name within the server's file system. The user selects an object based on its user-visible name, and the client retrieves it by constructing a handle from the server's host name and port number and the object's path name. Gopher's "search servers" maintain full-text inverted indices of subsets of the documents stored in a gopher server. They return to the client handles to documents that match a Boolean search pattern. Gopher clients can retrieve objects from archie and WAIS.

3.4 World-Wide Web

The World-Wide Web, or WWW [4], organizes data into a distributed hypertext, where nodes are full-text objects, directory objects called

1. Actually, the gopher information space is a directed graph, since it allows cycles.

cover pages, and indices. WWW also supports full text search over documents stored at a particular WWW server.

The WWW client provides users with a hypertext browsing interface. Besides its native *HyperText Transfer Protocol,* WWW clients understand FTP archives, gopher servers, archie, and the network news transfer protocol NNTP. HTTP allows document retrieval and full-text searches. HTTP objects are identified by their protocol type, the corresponding server's name, and the path name of the file where the directory or full-text object is stored.

3.5 Prospero

The Prospero File System [12] lets users build customized views, or *virtual systems,* of directories available throughout the Internet.

Views are essentially directories composed from various sources including a user's own views and imported views from other users. A user organizes his information space hierarchically, just like traditional file systems. When a user finds an interesting object, the user can include it in his view by linking to it. The object may be a simple file or a view. One special type of view is an index, which returns a directory of objects that satisfy some query. A view can be composed by applying a filter against an existing view. Filters customize the target view by reorganizing or extracting parts of it. Listing a Prospero view requires a computation distributed across all of the nodes reachable by transitive closure of all of the view's links, indices and filters.

Users advertise information by registering their virtual system with the Prospero server administrator. The administrator creates a link to the new virtual system in the master view of virtual systems, where other users can see and navigate through it.

3.6 Other Discovery Tools

Besides the services described above, other prototyped discovery tools include netfind, Semantic File Systems, and Nomenclator.[2]

Netfind [16] is a white-pages directory tool, which, given an Internet user's name and organization, tries to locate available information

2. We would like suggestions from the editor on whether we should delete or shorten this section.

about the user. A successful netfind query returns information like the user's e-mail address and phone number. Netfind builds its indexing database based on data scattered across a number of sources, such as netnews messages, the Domain Name System, the Simple Mail Transfer Protocol, and the *finger* utility. Among its advantages, we should point out that netfind relies only on already available information and mechanisms. However, *netfind* generates an average of 133 packets per second.

The Semantic File System [6] integrates associative access into a traditional tree-structured file system. Associative access is achieved by providing file systems with an attribute extraction and query interface. Attribute extraction is performed by filters called *transducers*. Queries consist of boolean combinations of the desired attribute-value pairs. Transducers and queries produce customized views of the file system hierarchy called *virtual directories,* which help locate and organize information. A semantic file system research prototype has been implemented on top of Sun NFS.

Nomenclator [14] implements attribute-based or *yellow-page* naming on top of hierarchical, white-page naming systems. A Nomenclator *access function* is, in essence, a server that periodically traverses other access functions and appropriate parts of the white-page name space, and constructs an index of the objects encountered that satisfy certain properties. The Nomenclator client uses a directory of services called the *active catalog* to identify access functions pertinent to a user's query.

4. Indexing Other Services

Indie can index the contents of non-Indie servers in two ways, depending on whether or not they actively cooperate. Non-Indie servers actively cooperate by calling the appropriate routines from Indie's gateway library. If they do not cooperate, Indie can index them by brute force. We explore these approaches below.

As Figure 6 illustrates, both approaches require an Indie broker modified to communicate with the non-Indie server. This broker serves as a gateway for the non-Indie server. Gateways register themselves with Indie's directory of services to be visible to Indie brokers. To the rest of Indie, a gateway broker is indistinguishable from an In-

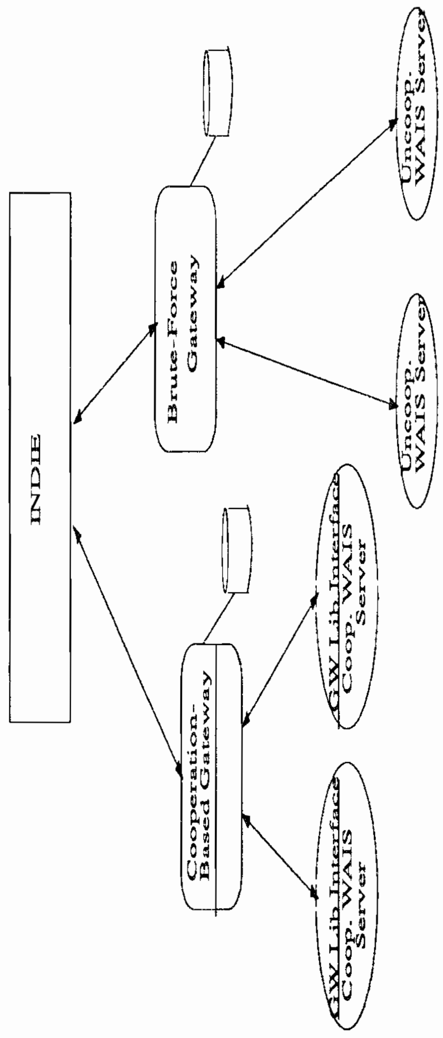
INDIE
Brute-Force Gateway
Uncoop. WAIS Server
Uncoop. WAIS Server
Cooperation-Based Gateway
GW Lib Interface Coop. WAIS Server
GW Lib Interface Coop. WAIS Server

Figure 6: Two Approaches to Interoperability.

die broker. They forward additions and deletions, accept generator objects in their trigger table, and register themselves with the directory of services.

Indie tries to preserve the autonomy of information providers by not giving ownership of their data away. Consequently, gateways to non-Indie servers describe the contents of objects available from the underlying servers, but do not contain the actual document.

4.1 Cooperative Interoperability

Cooperative non-Indie servers call the gateway library routines to register themselves with their gateway, create Indie object descriptors for each of their objects, and forward these to their gateway. They maintain consistency by informing their gateway of new, modified, and deleted objects.

Gateways periodically poll their non-Indie servers for updates. Non-Indie servers create and timestamp Indie object descriptors for their objects, and return changes to these object descriptors when their gateway polls them. The consistency maintenance mechanism between gateways and non-Indie servers is similar to the one between brokers. When polling a non-Indie server, the gateway sends the timestamp from its registration table entry associated with that server as part of the poll request. The server returns a list of Indie descriptors of objects whose timestamps are more recent. The gateway updates its registration table with the timestamp of the latest update. Recovery from failure or partition also relies on timestamps, and is similar to the media recovery mechanism executed between peer brokers.

When a client requests the contents of a specific object stored in a non-Indie server, the corresponding gateway retrieves the object from the server and forwards it to the client.

4.1.1 Gateway Library Primitives

The gateway library consists of the following (principle) primitives:

Register (gateway, server) Registers a non-Indie *server* with its *gateway*. This call is issued by *server,* and causes its description to be included in the *gateway's* registration table.

Unregister (gateway, server) Removes *server* from the *gateway's* registration table. This call also causes the corresponding object descriptors to be deleted from the *gateway's* database.

Oid [ ] =Poll (gateway, server, timestamp) Used by the *gateway* to poll a non-Indie *server* for updates more recent than *timestamp*. Servers respond with a sequence of Add() and Delete() calls.

Add (gateway, object descriptor) &

Delete (gateway, object descriptor) Adds or deletes *object descriptor* from the *gateway's* database.

Get_Contents (gateway, server, object) Used by the *gateway* to ask *server* for the contents of a particular *object*.

Send_Contents (gateway, object, contents) Sends the actual *contents* of an *object* to its *gateway*.

4.1.2 FTP Archives—A Special Case

Indie's interfaces to FTP archives is a special case of a cooperative approach. FTP site administrators construct a file of object descriptors for the directories or files they want to be visible to Indie (The descriptor file looks like a LaTex bib file). The FTP site administrator sends the gateway's administrator the host name of the FTP archive and the name of the descriptor file. The gateway's administrator enters this information in the gateway's registration table (This procedure is equivalent to the `register()` primitive mentioned above).

The gateway periodically spawns a process that polls the FTP archives listed in its registration table, retrieves each site's descriptor file, and compares it to a corresponding shadow version kept locally. It detects additions, deletions, and modifications to object descriptors in the descriptor files, and calls the gateway library to update the gateway accordingly. The gateway propagates appropriate changes to Indie brokers registered with it (once residency times of new objects are satisfied).

Indie brokers use this technique to load local data into their databases (see the object_resides_locally value of the `who_told_us` field in Figure 2).

4.2 Indexing by Brute Force

If a non-Indie Server does not cooperate, the Indie gateway must periodically extract indexing information from the server using the server's own protocols. Depending on the service, they do this by either querying the server for new objects, querying the server for objects that match keywords, or essentially performing a depth first search of the server's information space. The gateway must assign Indie object descriptors to these objects, identify additions, deletions, and changes to the objects retrieved since the last poll, and apply the differences to its database. Besides this brute force work, the gateway looks like any other Indie broker.

Brute force interoperability is less efficient than active cooperation, but requires no modifications from existing servers. We believe that services like WAIS and archie can be indexed by brute force, and gopher and WWW servers can cooperate by implementing root level "search servers". Of course, brute force polling should be infrequent enough that the server and the communication network are not overloaded.

5. Implementation Details

This section overviews our implementation of Indie, identifies incomplete components of the system, and describes support for the directory of service's ranking function.

We built Indie atop a real memory database and set of communication primitives called *DHT* that were designed for an ongoing distributed hypertext project here at USC [13]. We chose DHT because its creator agreed to support it rather than for reasons of design or functionality. However, because Indie's database and interprocess communication calls go through less than twenty DHT primitives, we can easily incorporate a different storage manager and communication library in future versions of Indie.

Indie's client user interface and broker administrator tool feel similar to *gopher's* terminal interface. Both tools are implemented with the UNIX Curses library. The client tool displays both text and Postscript, and understands UNIX directory listings and tar and compressed file formats. The client tool looks to the user's environment variables when selecting the pager program, text editor, and Postscript previewer. The client includes a special mode for displaying hierarchical directory listings conveniently (This mode is similar to Apple Macintosh's System 7's directory listing).

5.1 Ranking Function

Indie brokers and the directory of services currently employ a simple tagged, inverted index of attribute values of object descriptors when determining the set of objects and generators that 'hit'. We also incorporated a *subject index* that we believe may help rank these hits.

Librarians classify books with the Dewey Decimal System, the Library of Congress Classification System, and the ACM's category and subject descriptors. All of these schemes are hierarchically structured. Figure 7 illustrates how we exploit the Library of Congress (or any other) classification system when ranking brokers relevant to a user's query.

A broker's generator object can contain a list of Library of Congress ranges that describe the broker's desired contents. Figure 7 illustrates the Library of Congress ranges of three generators **G1**, **G2**, and **G3**. **G1**, a generator for a gateway to a university library, has range from A to Z. **G2**, a generator for a broker of math and astronomy books, ranges from QA to QC. **G3**, a generator for a broker on computer networks, has the two ranges QA76.6–QA77 and TK5000–TK8000. The directory of services orders hits by the degree of overlap between the query and the broker's Library of Congress ranges. For example, the query for QA76.64 returns the ordered list G3, G2, G1. Brokers that do not list their ranges are ranked last.

Indie employs a data structure called a *segment tree* [2] to implement this index. Using the subject index requires that the creators of Indie brokers and gateways attach Library of Congress ranges to their generators, and requires that users or their client programs tag their queries with example Library of Congress numbers.

5.2 Future Work

At the time this was written, we are implementing a cooperative gateway to WAIS, but currently use the gateway to FTP archives as the principle mechanism to load Indie with data. The gateway to WAIS

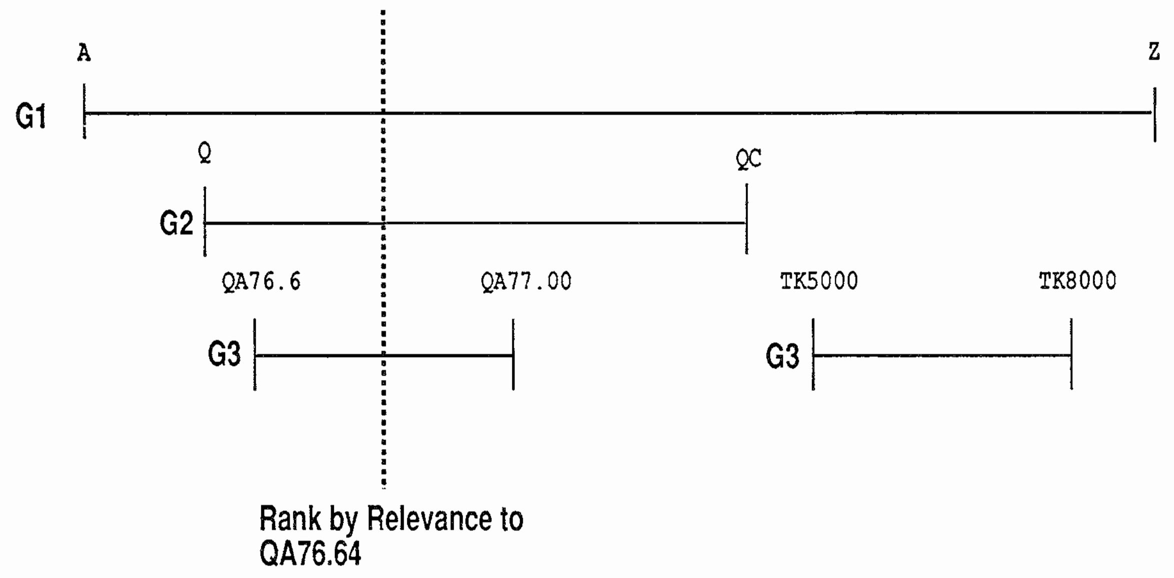

Figure 7: Ranking function based on overlap in classification.

uses the gateway library primitives, described above. Constructing the gateway requires creating an Indie broker and carefully modifying the WAIS server to call the gateway library in the appropriate place.

We have yet to test high degrees of replication for the directory of services, stress Indie's DHT database with tens of thousands of items, or gain sufficient experience with the gateway library. We have immediate plans to build brute foce gateways to gopher, WWW, and the USC library system. Given the gateway to the USC library, we will be able to evaluate the usefulness of Indie's subject index. In the long term, we would like to modify some existing X-window client into our client and administrator tools.

6. *Conclusions*

Indie exploits the database notion of triggers and rules to build an architecture that automatically clusters pointers to related information. Similar to the way that network news reading programs and Gifford's news clipping service [8] subscribe to particular data sources, the manager of an Indie broker registers its interest with other applicable brokers and gateways. Indie further exploits its rules and timestamped based consistency algorithm to replicate data.

The Internet community widely uses half a dozen resource discovery tools. Why do we need yet another? Indie offers functionality that exists no where else. No one has yet addressed building a directory of (discovery) services that all tools can use. Except for university research prototypes, no tool automatically clusters or indexes related information from distributed repositories, and none do it as efficiently as Indie. Users of distributed hypertext tools like WWW, Prospero, and gopher need a service like Indie to identify documents to which they should potentially link.

It is our hope that the concepts that motivated Indie will find their way into other discovery systems.

Glossary

- Indie Broker: A database of references to objects stored elsewhere.

- Directory of Services: The replicated database that stores the generator objects of all Indie brokers and gateways.
- Generator Object: The description of an Indie broker or gateway.
- Registration Table: Maintained by each Indie broker. Each entry describes an Indie broker or gateway to which this broker may subscribe.
- Trigger Table: Maintained by each Indie broker and gateway. Contains the generator object of brokers that index this service.
- Non-Indie Service: Refers to any discovery service or primary source of data such as a file, an FTP archive, a CD/ROM database, or a library catalogue.
- DHT: Distributed Hypertext Protocol used between system components.
- Gateways: Indie brokers dedicated to interface to non-Indie servers.
- Peer: If a broker indexes another broker, both are peers.

References

[1] R. Alberti, F. Anklesaria, P. Lindner, M. McCahill, and D. Torrey. The Internet Gopher protocol: a distributed document search and retrieval protocol. On-line documentation, Spring 1992.

[2] Jon Louis Bentley and Derick Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers,* 29(7):571–577, July, 1980.

[3] T. Berners-Lee, R. Cailliau, J-F. Groff, and B. Pollermann. World-Wide Web: An information infrastructure for high-energy physics. *Proceedings of the Workshop on Software Engineering, Artificial Intelligence and Expert Systems for High Energy and Nuclear Physics,* January 1992.

[4] T. Berners-Lee, R. Cailliau, J-F. Groff, and B. Pollermann. World-Wide Web: The information universe. *Electronic Networking: Research, Applications and Policy,* 1(2), Spring 1992.

[5] Peter B. Danzig, Jong Suk Ahn, John Noll, and Katia Obraczka. Distributed indexing: A technique for scalable, distributed information retrieval systems. *ACM SIGIR 91,* pages 220–229, March, 1991.

[6] Mark A. Sheldon, David K. Gifford, Pierre Jouvelot and James W. O'Tolle Jr. Semantic file systems. *Proceedings of the 13th ACM Symposium on Operating Systems Principles,* pages 16–25, October 1991.

[7] Alan Emtage and Peter Deutsch. archie: An electronic directory service for the internet. *Proceedings of the Winter 1992 Usenix Conference,* January 1992.

[8] David K. Gifford, Robert W. Baldwin, Stephen T. Berlin, and John M. Lucassen. An architecture for large scale information systems. *Proceedings of the 10th ACM Symposium on Operating Systems Principles,* 19(5):161–171, December 1985.

[9] Brewster Khale and Art Medlar. An information system for corporate users: Wide area information servers. *ConneXions—The Interoperability Report,* 5(11), November 1991.

[10] Clifford A. Lynch. The Z39–50 information retrieval protocol: An overview and status report. *ACM Computer Communication Review,* 21(1):58–70, 1991.

[11] B. Clifford Neuman. The virtual system model: A scalable approach to organizing large systems: A thesis proposal. Technical Report TR-90-05-01, University of Washington, Seattle, May 1990.

[12] B. Clifford Neuman. Prospero: A tool for organizing internet resources. *Electronic Networking: Research, Applications and Policy,* 2(1), Spring 1992.

[13] John Noll and Walt Scacchi. Integrating diverse information repositories: A distributed hypertext approach. *IEEE Computer,* 24(12):38–45, December 1991.

[14] Joan J. Ordille and Barton P. Miller. Nomenclator descriptive query optimization for large X.500 environments. *ACM SIGCOMM 91 Conference,* pages 185–196, September 1991.

[15] M. F. Schwartz, D. R. Hardy, W. K. Heinzman, and G. Hirschowitz. Supporting resource discovery among public internet archives using a spectrum of information quality. Technical Report Technical Report CU-CS-487-90, Department of Computer Science, University of Colorado, Boulder, Colorado, September 1990.

[16] Michael F. Schwartz. Experience with a semantically cognizant internet white pages directory tool. *Journal of Internetworking Research and Experience,* 1(2), December 1990.

[17] Michael Stonebraker and Lawrence A. Rowe et al. The postgres papers. Technical report, UC Berkeley, June 25, 1987.

[submitted June 25, 1992; accepted Aug. 6, 1992]