Peer-to-Peer I/O (P2PIO) Protocol Specification Version 0.6

K. Berket, A. Essiari, D. Gunter, W. Hoschek
Distributed Systems Department
Lawrence Berkeley Laboratory
1 Cyclotron Road, Berkeley, CA 94720
{kberket, aessiari, dkgunter, whoschek}@lbl.gov

Abstract

Today's distributed systems require simple and powerful resource discovery queries in a dynamic environment consisting of a large number of resources spanning many autonomous administrative domains. The distributed search problem is hard due to the variety of query types, the number of resources and their autonomous, partitioned and dynamic nature. We propose a generalized resource discovery framework that is built around an application level messaging protocol called Peer-to-Peer I/O (P2PIO). P2PIO addresses a number of scalability problems in a general way. It provides flexible and uniform transport-independent resource discovery mechanisms to reduce both the client and network burden in multi-hop P2P systems.

1 Introduction

Modern scientific research is conducted by large distributed multi-disciplinary teams that are cooperating to conduct experiments, simulations, and achieve results. Such collaborations are often globally distributed and multi-institutional. A difficult problem for these teams is often finding and organizing data, results, and resources. Many collaborations solve this problem by defining centralized storage and managing the resources as a single domain. But, this solution scales poorly and does not allow for opportunistic use of resources and data repositories. A much more scalable alternative would be to leverage the ideas of Peer-to-Peer computing to build a Peer-to-Peer resource discovery framework.

Distributed applications used in modern science are characterized by large scale, heterogeneity, lack of central control, multiple autonomous administrative domains, unreliable components and frequent dynamic change. In such applications it is desirable to maintain and query dynamic information about active participants such as services, resources and user communities in a timely manner. Peer-to-Peer file sharing systems, instant messaging services, moni-

toring infrastructures, resource brokers, job schedulers and flexible bootstrap configuration systems all share these requirements.

For example, physics science users would like to easily share information such as Grid input and output files with collaborators, independent of the churn rate of the collaboration. For good decision-making, a resource broker service would like to find the available job execution services, their capacity and utilization. Further, it needs to find the storage services that have given input files, the storage services that can store output files for the user, as well as find the access control policies and bandwidth between all involved services.

Administrators and performance analysts would like to find and display the average latency and bandwidth between all or a subset of Grid nodes over the last 24-hour period. In case of an environmental or epidemic disaster, a future emergency response team would like to quickly mine the vast data of a large variety of active sensors (wind, earth vibrations, highway traffic), and passive databases (historic sensor data, population statistics, immunization records). The use cases above share many commonalities, but their current implementations differ in that they utilize a variety of heterogeneous database technologies, data formats, query and response languages, routing strategies and network protocols. They also exhibit different degrees of node churn: some participants are almost always available (e.g. core servers), while others frequently come and go (e.g. laptop users).

The distributed search problem is hard due to the number of resources and their dynamic nature. The number of resources and their churn rate is steadily increasing. A number of existing P2P solutions are specific to a problem domain (e.g. Gnutella [2], Freenet [3], Tapestry [4], Chord [5], Globe [6], JXTA [7, 8, 9]) and are not applicable to a more general system. More centralized solutions, such as RDBMS [10], UDDI [11], GMA [12, 13], ANSA and CORBA [14, 15] also offer limited solutions that are specific to a domain. However, they do not scale well to a large dynamic system. Lookup Services such as LDAP and MDS [16, 17], Jini [18], SLP [19, 20], SDS [21] and INS [22] also do not scale well, or are not general enough. NEED TO REWRITE THIS PARAGRAPH.

We propose a generalized resource discovery framework that is built around an application level messaging protocol called Peer-to-Peer I/O (P2PIO), derived from our previous Peer Database Protocol (PDP) [23]. P2PIO addresses a number of scalability problems in a general way. It provides flexible and uniform transport-independent resource discovery mechanisms to reduce both the client and network burden in multi-hop P2P systems.

The P2PIO messaging protocol is transport independent with a well-defined set of messages that can be used to support a variety of use cases. P2PIO addresses a number of scalability problems. Its flexible and uniform transport-independent resource discovery mechanisms can reduce both the client and the network burden in a multi-hop P2P system. It provides effective and scalable I/O abstractions which allow for incremental queries that can gather results sequentially rather than all at once. This is similar to the sequential file I/O model. P2PIO can also be seen as extending traditional distributed database I/O and query mechanisms to a peer-to-peer setting. To support a large vari-

ety of use cases, P2PIO is designed to support any query language, including XQuery, SQL, XPath and regular expressions. By providing routed and direct response mode [1], it supports different models of control, efficiency and security. Arbitrary scope and neighbor selection policies can be specified to enable smart dynamic routing. Such policies allow a client to select the data and peers the query should be applied to. The communication protocol is transport independent to be useful in various application environments, and hence can be implemented over TCP, multicast, SOAP, etc. An implementation over SOAP/HTTP(S) is particularly important for interoperability in heterogeneous environments, and for integration into existing frameworks and established Grid Web Service standards such as OGSI/WSRF. Efforts to create a Global Grid Forum P2P resource discovery research group based on P2PIO are currently ongoing.

A P2P I/O mechanism should provide familiar, effective and scalable I/O abstractions to P2P networks. Traditional sequential, stateful, segmented file I/O, both in synchronous and asynchronous manner provides such mechanisms: A very large file does not need be read into memory in one large read; rather, it is better sequentially read in segments of N bytes each. In asynchronous mode, applications are notified whenever a byte segment of a (remote) file becomes available. Hence, an important feature of P2PIO is that it is designed to enable incremental discovery by extending these familiar, effective and scalable I/O mechanisms to the P2P environment: A client opens a transaction that specifies a query. It then iterates over the items of the query result set. It reads a segment of N items from the result set, processes it, then reads the next segment of N items, and so on. After reading parts of or all of potentially millions of items, the client closes the transaction, indicating to the P2P network that it may release state and resources associated with the transaction. Items may be read either in synchronous style (single mode, e.g. polling catalog browsing) or in asynchronous style (multi mode, e.g. pushing event notifications such as monitoring updates). In single mode the client retains control over flow and resource consumption whereas in multi mode the client effectively transfers such control to the server. Similarly, opening a transaction can be done in synchronous or asynchronous manner, using invitations. Coarse grained high-level abstractions (e.g. reading all items of a query) can be layered on top of these fine grained low-level abstractions.

The abstractions that P2PIO can support are quite powerful. In particular, the use cases outlined in the motivation can be implemented by application-specific plug-ins once the framework is in place. For example, the resource broker use case could use an XQuery language module, in conjunction with an XML module, flooding as a routing strategy, a simple XPath query for neighbor selection, and a topology adaptation algorithm directed towards high availability.

The rest of this paper is organized as follows. We first discuss the problem in further detail and present existing systems. Then, we discuss the specific goals of the P2PIO protocol and introduce the mechanisms that solve these problems. We then discuss the P2PIO messaging model, followed by the message

specifications.

2 Overview (need to rewrite)

The P2PIO protocol is an application level messaging protocol for generalized resource discovery. It addresses a number of scalability issues at the client and network level. In this section we discuss these issues and the mechanisms provided by P2PIO to address them.

2.1 General Motivation

In many systems querying is accomplished by filtering at the client. This is inefficient as the client may have to receive a large number of results and perform substantial processing. The P2PIO protocol provides support for efficient handling of simple and medium queries, and can also support complex queries. The client is given the ability to have the result set be routed through the network. This allows for filtering to be applied in a distributed manner and reduces the burden on the client.

In use cases regarding resource discovery, a client often issues a query that generates a large number of results. In many instances the client is satisfied by the first couple of responses and disregards the remainder. This scenario wastes resources at the client and in the network. P2PIO introduces a mechanism that the client can use to avoid this scenario. The client is given the ability to iterate through the result set. This allows the client to control the number of incoming results.

Efficient query routing in a resource discovery system increases the scalability of the system. The P2PIO protocol provides support for efficient smart dynamic query routing by allowing the client to specify a neighbor selection query as part of its request. NEED MORE TEXT HERE.

Routing of the result set through the network is not efficient if intermediate processing is not necessary, as in the case of simple queries. Here, it is more efficient to deliver the result set directly to the client. The P2PIO protocol allows the client to have the result set sent directly to the client. In order to alleviate the load on the client in this mode, an invitation is sent to the client by the data provider prior to sending the result set.

2.2 Familiar, Effective and Scalable I/O Abstractions

A P2P I/O mechanism should provide familiar, effective and scalable I/O abstractions to P2P networks, along the lines of sequential, stateful, segmented file I/O, both in synchronous and asynchronous manner.

Here, a client opens a file, and is given a unique file handle. Using the file handle as context, it then iterates over the file. It reads a segment of N bytes, processes it, then reads the next segment of N bytes, and so on. After reading parts or all of the file, the client closes the file, indicating to the file system that

it may release state and resources associated with the file handle. Typically, the next N bytes may be read in synchronous style (e.g. polling file browsing) or in asynchronous style (e.g. pushing monitoring updates). Similarly, opening a file can be done in synchronous or asynchronous manner. Coarse grained high-level APIs (e.g. reading *all* bytes of the file into a string) can be layered on top of these fine grained low-level abstractions.

In analogy to traditional sequential, stateful, segmented file I/O, a P2PIO client opens a transaction that specifies a query, and is given a unique transaction identifier. Using the transaction identifier as context, it then iterates over the items of the query result set. It reads a segment of N items from the result set, processes it, then reads the next segment of N items, and so on. After reading parts or all of the items, the client closes the transaction, indicating to the P2P network that it may release state and resources associated with the transaction. The next N items may be read in synchronous style (*single mode*, e.g. polling catalog browsing) or in asynchronous style (*multi mode*, e.g. pushing monitoring updates). In single mode the client retains control over flow and resource consumption whereas in multi mode the client effectively transfers such control to the server. Similarly, opening a transaction can be done in synchronous or asynchronous manner, using invitations. Coarse grained high-level APIs (e.g. reading *all* items of a query into a set) can be layered on top of these fine grained low-level abstractions.

Besides extending file I/O like mechanisms to the P2P case, P2PIO can also be seen as extending traditional distributed database I/O and query mechanisms to the P2P case.

2.3 Simple to Complex Queries

A P2P network can be efficient in answering queries that are recursively partitionable [23]. Here, query processing can be parallelized and spread over all participating nodes. Potentially very large amounts of information can be searched while investing little resources such as processing time per individual node. The recursive parallel spread of load implied by a recursively partitionable query is the basis of the massive P2P scalability potential. However, query performance is not necessarily good, for example due to high network I/O costs. Example queries are as follows:

- *Simple Query (recursively partitionable): Find all (available) services that have download capacity greater than 10 MB/s.*
- *Medium Query (recursively partitionable): Return the number of replica catalog services. Find the service with the largest uptime.*
- *Complex Query (not recursively partitionable): Find all (execution service, storage service) pairs where both services of a pair live within the same network domain.*

To support a large variety of use cases, P2PIO is designed to support all given query types. P2PIO is an extensible protocol framework. Arbitrary queries in

arbitrary query languages can be posed, returning arbitrary items as query results. For example, we have prototyped query plugins for XQuery, SQL, XPath and regular expressions.

2.4 Multiple Response Modes

A P2P I/O mechanism should support use cases requiring different levels of control, efficiency, security, trust and protection against DOS attacks, leading to routed and direct response mode [23]. Under *Routed Response*, items are fanned back into the originator along the paths on which the query flowed outwards. Each server returns to its client not only its own local items but also all remote items it receives from neighbors. Under *Direct Response*, items are not returned by routing through intermediary peers. Each server that has local items directly invites the agent to retrieve items, which the agent then combines and hands back to the originator.

Neigher response mode is ideal for all use cases. Hence, P2PIO is designed to support both response modes.

2.5 Smart Dynamic Routing

To enable smart dynamic routing [24], arbitrary scope and neighbor selection policies can be specified. Such policies allow a client to select the data and peers the query should be applied to. NEED MORE HERE.

2.6 Reliability etc.

The protocol is transport independent to be useful in various environments, and hence can be implemented over TCP, multicast, SOAP, etc. For improved reliability message timeouts are used, and persistent transport connections are not required. This also helps in dealing with heterogeneous peer capabilities and capacities. NEED MORE HERE.

3 P2PIO Messaging Model

3.1 Transactions

We consider a *network of peer* processes. Each peer forwards, sends, receives and processes messages according to this specification. All message exchanges occur within the context of a transaction. The specification of the entire P2P network is given by defining the individual transactional message exchanges between any two peers.

A *transaction* is a sequence of one or more message exchanges between two peers for a given query. This non-trivial transaction model is in contrast to a simpler model where a transaction consists of a single request-response message exchange (e.g. HTTP). The former model is stateful whereas the latter is stateless. A peer can concurrently handle multiple independent transactions.

A process that initiates a transaction is called an *originator*. The originator may not necessarily be a part of the network of peers, but it must interact with a peer as its entry point into the system. The peer that the originator interacts with is called the *agent* for this transaction. The agent in turn interacts with the rest of the P2P network on behalf of the originator. The interactions between the originator and agent MAY be specified on a per application basis, and thus are not discussed in this specification.

When a pair of peers is involved in a message exchange, the initiator of the exchange is referred to as the *requestor* or *client*, and the other peer is referred to as the *responder* or *server*. When an individual message is discussed we will refer to the peer that sent that message as the *source* and the peer that (potentially) receives the message as the *destination*.

A transaction is identified by a transaction identifier. All messages of a given transaction MUST carry the same transaction identifier. The transaction identifier is a UUID [25] that SHOULD be globally unique for the entire lifetime of the transaction. In practice, it is sufficient for the UUID to be unique with exceedingly large probability, suggesting the use of a 128 bit integer computed by a cryptographic hash digest function such as MD5 [26] or SHA-1 [27] over originator IP address, current time and a random number.

3.2 Message Exchanges

The P2PIO messaging model employs four request messages (OPEN, RECEIVE, INVITE, CLOSE) and four response messages (SEND, FINALSEND, OK, ERROR).

OPEN. An OPEN message along with its query and timeout is used to initialize a transaction through which items matching the query may subsequently be retrieved. A peer accepting an OPEN message MUST respond with an OK or ERROR message before forwarding the OPEN message to its dependents. The message MAY be forwarded along peer hops through the P2P topology, in which case the transaction identifier MUST stay invariant across hops. Items are explicitly requested via a subsequent RECEIVE message. An OPEN message contains the opaque query itself, a timeout, a response mode indicator (see below), as well as a transaction identifier that MUST be chosen by the originator.

RECEIVE, SEND and FINALSEND. A RECEIVE message is used by a client to request remaining query items from a server. It requests that the server SHOULD return a set of at least `min` and at most `max` items from the (remainder of the) item set. This corresponds to the `next(N)` method of an iterator (operator). For example, a low latency use case can use `min=1, max=10` to indicate that at least one and at most ten items SHOULD be delivered in response to the RECEIVE request.

A RECEIVE request contains a parameter that asks for delivery of items in either single (pull) or multi (push) mode, restricting the number of messages

that may be sent in response to this RECEIVE message. In *single mode* a single RECEIVE request MUST precede every single FINALSEND response. An example sequence is RECEIVE-FINALSEND-RECEIVE-FINALSEND. In *multi mode* a single RECEIVE request asks for a sequence of zero or more SEND messages followed by a FINALSEND message. A client need not explicitly request more items, as they are automatically pushed. An example sequence is RECEIVE-SEND-SEND-FINALSEND-RECEIVE-SEND-FINALSEND.

A FINALSEND message indicates the end of the current RECEIVE exchange. The FINALSEND message MAY optionally contain a parameter marking the transaction as "still open". If this parameter is not present it indicates that the transaction is closed due too item set exhaustion. A FINALSEND marked as "still open" MAY also optionally contain the number of remaining items currently locally available for immediate delivery in response to a subsequent RECEIVE. A FINALSEND marked as "still open" MAY also optionally contain the current estimated number of globally remaining total items eventually available for delivery in response to a subsequent RECEIVE. A client can successively issue RECEIVE messages until the item set is exhausted. A client need not retrieve all items from the entire item set. For example, after having received the first 10 items it MAY issue a CLOSE request.

CLOSE. A client may issue a CLOSE message to close the transaction, informing the server that the remaining items (if any) are no longer needed and can safely be discarded.

OK and ERROR. An OK message is used to positively acknowledge a message. An ERROR message is used to negatively acknowledge a message.

INVITE. An INVITE message is used to indicate that, in response to a direct response OPEN message, the source of the INVITE has items available. INVITE messages only apply to Direct Response mode (see below).

Message Exchange Patterns. A pair of peers interacts by exchanging messages. Discrete messages belong to well-defined message exchange patterns. For example, the pattern of synchronous exchanges (one-to-one, pull) is supported as well as the pattern of asynchronous exchanges (one-to-many, push). The following message exchanges are permitted:

```
OPEN    --> OK | ERROR
RECEIVE --> (SEND [0:N], FINALSEND) | ERROR
INVITE  --> OK | ERROR
CLOSE   --> OK | ERROR
```

For example, a server responds to a RECEIVE message with an ERROR message or zero or more SEND messages followed by a FINALSEND message. Another example message exchange is OPEN-OK. An example transaction is a OPEN-RECEIVE-FINALSEND-RECEIVE-FINALSEND-CLOSE message sequence.

Routed Single	Routed Multi	Direct Single	Direct Multi
`--> OPEN`	`--> OPEN`	`--> OPEN`	`--> OPEN`
`<-- OK`	`<-- OK`	`<-- OK`	`<-- OK`
`--> RECEIVE`	`--> RECEIVE`		
`<-- FINALSEND`	`<-- SEND`	`<-- INVITE`	`<-- INVITE`
`--> RECEIVE`	`<-- FINALSEND`	`--> OK`	`--> OK`
`<-- FINALSEND`	`--> RECEIVE`	`--> RECEIVE`	`--> RECEIVE`
`--> CLOSE`	`<-- SEND`	`<-- FINALSEND`	`<-- SEND`
`<-- OK`	`<-- FINALSEND`	`--> RECEIVE`	`<-- FINALSEND`
	`--> CLOSE`	`<-- FINALSEND`	`--> RECEIVE`
	`<-- OK`	`--> CLOSE`	`<-- SEND`
		`<-- OK`	`<-- FINALSEND`
			`--> CLOSE`
			`<-- OK`

Figure 1: Message flows from client to server (“`-->`”) and back (“`<--`”).

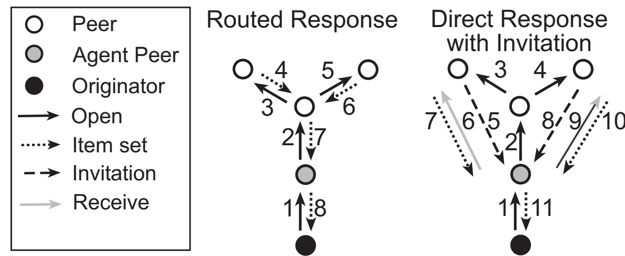

Figure 2: Routed and Direct Response Modes.

Response Modes. The message exchanges of a transaction depend on the *response mode* in use. Under *Routed Response*, items are fanned back into the originator along the paths on which the query flowed outwards. Under *Direct Response*, items are not returned by routing through intermediary peers. Each server that has local items directly invites the agent to retrieve items, which the agent then combines and hands back to the originator. Invitation occurs with an INVITE message that solicits a RECEIVE message. Interaction then proceeds with the normal RECEIVE-SEND-CLOSE pattern initiated by the agent. The messaging model is exemplified by the message flows depicted in Figure 1 and Figure 2.

A transaction in Routed Response mode consists of an OPEN exchange, followed by zero or more RECEIVE exchanges, followed by zero or one CLOSE exchange. In contrast, a transaction in Direct Response mode consists of an OPEN exchange from the agent, followed by one INVITE exchange from each responder to the agent (offering matching items for the query), followed by zero or more RECEIVE exchanges from the agent to each responder, followed by zero or one CLOSE exchange from the agent to each responder.

These response modes are summarized as follows:

```
Routed Response Mode: OPEN -->                 RECEIVE [0..N] --> CLOSE [0..1]
Direct Response Mode: OPEN --> INVITE[0..k] --> RECEIVE [0..N] --> CLOSE [0..k]
```

3.3 State Transitions

A peer maintains a state table. For each transaction (i.e. query) at least the transaction identifier, transaction timeout, and an open/closed state flag are kept. The identifier and timeout of a transaction are kept constant throughout the P2P network so that loops in message routes can be detected reliably. An example state table reads as follows:

Transaction Identifier	Transaction Timeout	State
100	20	Closed
200	50	Open

A transaction is *known* to a peer if the state table already holds a transaction identifier equal to the transaction identifier of the transaction. Otherwise, it is said to be *unknown.* A known transaction can be in two states: *open* or *closed.* The state transitions from *unknown* to *open* to *closed* and back to *unknown* state are depicted in Figure 3 and defined as follows:

- **Unknown to Open.** When an *unknown* transaction arrives with an OPEN message, it moves into *open* state. When a transaction moves into *open* state, it becomes known and MAY be forwarded to the neighbors obtained from neighbor selection.

- **Open to Closed.** A transaction moves from *open* into *closed* state when its transaction timeout has been reached, if the item set is exhausted by a FINALSEND, if a client issues a CLOSE to indicate that it is no longer interested in the (remainder of the) item set, or if one of several errors occur. Under direct response, a transaction to a non-agent peer *also* moves from *open* into *closed* state if the query produces no local items, or if it does produce local query items but the INVITE request is not accepted by the agent.

  When a transaction moves into *closed* state, a CLOSE request SHOULD be asynchronously forwarded to all dependents in order to inform them as well. A peer depends on a set of other peers (*dependents*) that are involved in item set delivery. Under Routed Response, the dependants are the peers obtained from neighbor selection. Under Direct Response, the dependents of an agent are the peers from which the agent has accepted an INVITE message, whereas all other peers have no dependents.

- **Closed to Unknown.** A transaction moves from *closed* state into *unknown* state when its transaction timeout has been reached. In other words, the transaction SHOULD be deleted from the state table.

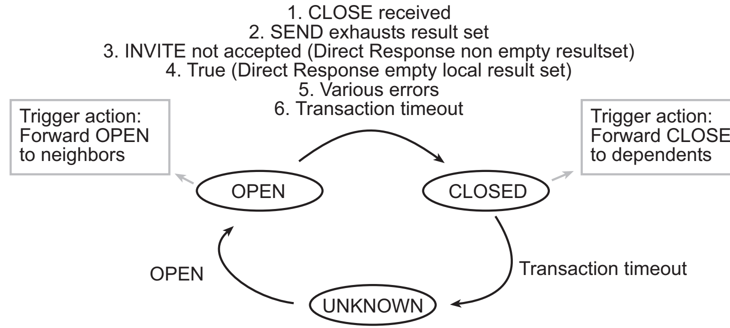

Figure 3: Peer State Transitions.

Message Acceptance and Rejection. An OPEN request MAY be accepted if the transaction is *unknown*. If an already known OPEN arrives, this usually indicates loop detection [28, 29]. The message MUST be rejected with an ERROR (e.g. "transaction already in use"). When a message other than OPEN arrives that has an unknown transaction identifier, it MUST be rejected with an ERROR (e.g. "transaction unknown"). RECEIVE, SEND, FINALSEND, CLOSE and INVITE messages MAY be accepted for a transaction in *open* state. A message for a transaction in *closed* state MUST NOT be accepted; the response to such a message is always an ERROR (e.g. "transaction already closed").

4 Message Specifications

This section defines the detailed normative semantics and syntax of all individual messages. P2PIO uses straightforward XML representations specified by means of a W3C XML Schema [30]. These XML schemas are given in the appendix.

4.1 OPEN

An OPEN message along with its query is used to initialize a transaction through which items matching the query MAY subsequently be retrieved. An OPEN message contains the query itself as well as a transaction identifier. P2PIO considers a query as an opaque information payload. A query is an arbitrary application-defined XML fragment. Particular query schemas are commonly used for specific applications, but P2PIO does not require the use of a query schema or any schema language. The concrete type of query is uniquely identified by the namespace and element name in use.

An OPEN message also contains scope hints that directly or indirectly define the global input fed to the query. The scope contains a transaction timeout, a maximum hop count, an optional neighbor selection query [24] and, optionally, arbitrary additional data as an extension.

The transaction timeout specifies the absolute time (in milliseconds) at which this transaction SHOULD timeout and be implicitly closed. The destination MAY disagree with the given timeout, in which case it MUST respond with an ERROR message. Otherwise it MAY respond with an OK message, and MUST use the same timeout value in any forwarded OPEN messages.

The hop count indicates the maximum number of hops that this transaction SHOULD be forwarded. It MUST be greater than or equal to zero. A value of zero indicates that the query MUST NOT be forwarded anymore and hence be only applied to the local database of the server. The destination MUST reject a hop count less than zero as an ERROR and reduce the value by at least one in any forwarded OPEN messages.

The optional neighbor selection query specifies to which neighbors the OPEN message SHOULD be forwarded. If it is not given, the server MAY choose to use any neighbor selection policy it sees fit. A neighbor selection query is an arbitrary XML fragment, the type of which is uniquely identified by the namespace and element name in use.

The message also contains a hint indicating what response mode MUST be used (Routed Response or Direct Response). Under Direct Response, also a locator to one or more network endpoints of the agent peer MUST be included, indicating where and over which transport mechanism an INVITE request SHOULD be sent. This enables peers with matches to INVITE the agent to RECEIVE their respective item set. Endpoint order is from most preferred to least preferred. The inviting peer MUST use only a single such endpoint as a destination for its INVITE message. An endpoint is a URI but SHOULD in fact be a URL.

An example OPEN message is as follows:

```
<open xmlns="http://dsd.lbl.gov/p2pio-1.0">
  <transactionID>4f76-8d0a-40d81de79445</transactionID>
  <query>
    <firefish:p2pQuery mergeOperator="concat" xmlns:firefish="http:
        //dsd.lbl.gov/firefish-1.0">
      <firefish:dataSourceQuery>
        <firefish:xPathQuery>
          <firefish:expression>/*/*</firefish:expression>
          <firefish:namespaces/>
        </firefish:xPathQuery>
      </firefish:dataSourceQuery>
    </firefish:p2pQuery>
  </query>
  <scope>
    <timeout>2003-10-03T13:48:36.917-07:00</timeout>
    <maxHops>3</maxHops>
    <extension/>
  </scope>
  <responseMode>
    <routedResponseMode/>
  </responseMode>
</open>
```

4.2 RECEIVE

A RECEIVE message is used by a client to request query items from a server. It requests that the server SHOULD return a set of at least `min` and at most `max` items from the (remainder of the) item set. This corresponds to the `next()` method of an iterator (operator). We have `1` $\leq$ `min` $\leq$ `max`. For example, a low latency use case can use `min=1, max=10` to indicate that at least one and at most ten items SHOULD be delivered in response to this RECEIVE request. `min=max=infinity` indicates that all remaining items SHOULD be delivered.

A client can successively issue RECEIVE messages until the item set is exhausted. A client need not retrieve all items from the entire item set. For example, after having received the first 10 items it MAY issue a CLOSE request.

A RECEIVE request contains a parameter that asks for delivery of items in either single (pull) or multi (push) mode, restricting the number of messages that may be sent in response to this RECEIVE message. In *single mode* a single RECEIVE request MUST precede every single FINALSEND response. An example sequence is RECEIVE-FINALSEND-RECEIVE-FINALSEND. In *multi mode* a single RECEIVE request asks for a sequence of zero or more SEND messages followed by a FINALSEND message. A client need not explicitly request more items, as they are automatically pushed. An example sequence is RECEIVE-SEND-SEND-FINALSEND-RECEIVE-SEND-FINALSEND. A FINALSEND message indicates the end of the current RECEIVE exchange.

A RECEIVE request also has an optional timeout (in milli seconds) indicating the absolute time at which this RECEIVE request SHOULD expire. If the timeout is not specified, the server SHOULD set the timeout for this request to the timeout associated with this transaction. In any case, a server MAY choose its timeout as it sees fit, but the timeout MUST be less than or equal to the overall transaction timeout. Upon timeout any response messages from the server MAY be ignored by the client. Upon timeout, if the server has not yet responded with the FINALSEND of this request, the server MUST now return with a FINALSEND, containing whichever items it has available within the item number constraints specified by the client, possibly zero items.

An example RECEIVE message is as follows:

```
<receive mode="single" xmlns="http://dsd.lbl.gov/p2pio-1.0">
  <transactionID>4f76-8d0a-40d81de79445</transactionID>
  <min>100</min>
  <max>10000</max>
  <timeout>2003-10-03T13:48:38.199-07:00</timeout>
</receive>
```

4.3 SEND and FINALSEND

When a peer accepts a RECEIVE message, it responds with zero or more SEND messages followed by a FINALSEND message, in total containing `P` items from the (remainder of the) item set. Individual SEND and FINALSEND messages MUST each contain zero or more items. P2PIO considers an item as an opaque

information payload. An item is an arbitrary application-defined XML fragment. Particular item schemas are commonly used for specific applications, but P2PIO does not require the use of an item schema or any schema language.

A FINALSEND message indicates the end of the current RECEIVE exchange. We have `P` $\leq$ `max`. We MAY have `min` $\leq$ `P`. For example, zero or less than `min` items MAY be delivered when the entire query item set is exhausted, on timeout, or if the peer decides to override and decrease `P` for reasons including resource consumption control.

The FINALSEND message MAY optionally contain a parameter marking the transaction as "still open". If this parameter is not present it indicates that the transaction is closed due too item set exhaustion, in which case the server MUST respond with an ERROR to any subsequent RECEIVE message. If the parameter is present the client MAY issue a subsequent RECEIVE request.

A FINALSEND marked as "still open" MAY also optionally contain the number of remaining items currently locally available for immediate delivery in response to a subsequent RECEIVE (`locallyAvailable`). `locallyAvailable` MUST be greater than or equal to zero. `locallyAvailable=0` can indicate that remote peers have not yet delivered items necessary to return more than zero items for a subsequent RECEIVE. If `locallyAvailable` is not present, it indicates that the quantity is unknown.

A FINALSEND marked as "still open" MAY also optionally contain the current estimated number of globally remaining total items eventually available for delivery in response to a subsequent RECEIVE (`globallyAvailable`). The actual number of items that can (later) be delivered may be larger. It SHOULD not be smaller, except if other peers fail to deliver their suggested items. `globallyAvailable` MUST be greater than or equal to zero, and greater or equal to `locallyAvailable`, if present. If `globallyAvailable` is not present, it indicates that the quantity is unknown.

An example SEND message is as follows:

```
<send xmlns="http://dsd.lbl.gov/p2pio-1.0">
  <transactionID>4f76-8d0a-40d81de79445</transactionID>
  <itemSet>
    <article code="13563275" xmlns="">
      <headline_text>Version 1.0 XML Standard</headline_text>
      <source>W3C</source>
    </article>
    <article code="13560996" xmlns="">
      <headline_text>P2P Research Accelerating</headline_text>
      <source>BBC</source>
    </article>
  </itemSet>
</send>
```

An example FINALSEND message is as follows:

```
<finalSend xmlns="http://dsd.lbl.gov/p2pio-1.0">
  <transactionID>4f76-8d0a-40d81de79445</transactionID>
  <itemSet>
```

```
    <article code="13563275" xmlns="">
      <headline_text>Version 1.0 XML Standard</headline_text>
      <source>W3C</source>
    </article>
    <article code="13560996" xmlns="">
      <headline_text>P2P Research Accelerating</headline_text>
      <source>BBC</source>
    </article>
  </itemSet>
  <isStillOpen>
    <locallyAvailable>100</locallyAvailable>
    <globallyAvailable>1000000</globallyAvailable>
  </isStillOpen>
</finalSend>
```

4.4 CLOSE

A client MAY issue a CLOSE message to close the transaction, informing the server that the remaining items (if any) are no longer needed and can safely be discarded. The server MUST respond immediately with an OK or ERROR message. At the same time, the server MAY asynchronously forward the CLOSE to neighbors involved in item set delivery, which in turn MAY forward the CLOSE to their neighbors, and so on. Being informed of a CLOSE allows a server to release resources as early as possible. Strictly speaking, a client need not issue a CLOSE, and a server need not forward further a CLOSE, because a query eventually times out anyway. Even though this is considered misbehavior, a server must continue to operate reliably under such conditions.

An example CLOSE message is as follows:

```
<close xmlns="http://dsd.lbl.gov/p2pio-1.0">
  <transactionID>4f76-8d0a-40d81de79445</transactionID>
</close>
```

4.5 INVITE

An INVITE message is used to indicate that, in response to a direct response OPEN message, the source of the INVITE has items available. INVITE messages only apply to Direct Response mode. Here, a peer forwards the query to the peers obtained from neighbor selection without ever waiting for their item sets. The peer only applies the query to its local database. If the local item set is not empty, the peer directly contacts the agent with an INVITE message to solicit a RECEIVE message. Interaction then proceeds with the normal RECEIVE-SEND-CLOSE pattern, either in single (pull) or multi (push) manner (see above).

An INVITE message MUST contain a parameter marking the transaction as "still open". It MAY also optionally contain the number of remaining items currently locally available for immediate delivery in response to a subsequent RECEIVE (`locallyAvailable`). `locallyAvailable` MUST be greater than or

equal to zero. It MAY also optionally contain the current estimated number of globally remaining total items eventually available for delivery in response to a subsequent RECEIVE (`globallyAvailable`). The actual number of items that can (later) be delivered MAY be larger. `globallyAvailable` MUST be greater than or equal to zero, and greater or equal to `locallyAvailable`, if present.

An INVITE message MAY also include a locator to one or more network endpoints to indicate where and over which transport mechanism RECEIVE requests SHOULD be sent. Endpoint order is from most preferred to least preferred. An endpoint is a URI but SHOULD in fact be a URL. The destination MUST use only a single such endpoint for all RECEIVEs, and all endpoints MUST offer the same items for the given transaction. Hence within a single transaction, parallel striped transfers across endpoints are not permitted.

An example INVITE message is as follows:

```
<invite xmlns="http://dsd.lbl.gov/p2pio-1.0">
  <transactionID>4f76-8d0a-40d81de79445</transactionID>
  <locator>
    <endpoint>http://doggy.lbl.gov:8080/services/firefish</endpoint
        >
  </locator>
  <isStillOpen>
    <locallyAvailable>100</locallyAvailable>
    <globallyAvailable>1000000</globallyAvailable>
  </isStillOpen>
</invite>
```

4.6 OK

An OK message is used to positively acknowledge a message of a transaction. It contains no other information.

An example OK message is as follows:

```
<ok xmlns="http://dsd.lbl.gov/p2pio-1.0" >
  <transactionID>4f76-8d0a-40d81de79445</transactionID>
</ok>
```

4.7 ERROR

An ERROR message is used to negatively acknowledge a message. It indicates that some kind of error has occured within the transaction. An ERROR message contains a 3-digit integer status/error code used for programatic handling of errors, as well as a status reason giving a short textual human readable status/error description of the status code. Further, it MAY contain zero or more implementation dependent error cause strings (e.g. for stack traces).

An example ERROR message is as follows:

```
<error xmlns="http://dsd.lbl.gov/p2pio-1.0">
```

```
  <transactionID>4f76-8d0a-40d81de79445</transactionID>
  <code>557</code>
  <codeAsText>Transaction already closed</codeAsText>
</error>
```

5 Complete Normative XML Schema

The following are the complete normative W3C XML schema specifications for the messages described in this document. The definitions in this section MUST be considered normative, if there are any discrepancies between the definitions in this section and those portions described in other sections above.

Listing 1: Complete Normative W3C XML Schema for P2PIO Protocol

```
<!-- ############################################## -->
<!-- XML Schema specification of P2PIO protocol      -->
<!-- $Revision: 1.2 $, $Date: 2004/01/30 01:01:05 $  -->
<!-- ############################################## -->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:p2pio="http://dsd.lbl.gov/p2pio-1.0"
  targetNamespace="http://dsd.lbl.gov/p2pio-1.0"
  elementFormDefault="qualified"

  <xsd:element name="open" type="p2pio:OpenType"/>
  <xsd:element name="receive" type="p2pio:ReceiveType"/>
  <xsd:element name="send" type="p2pio:SendType"/>
  <xsd:element name="finalSend" type="p2pio:FinalSendType"/>
  <xsd:element name="close" type="p2pio:CloseType"/>
  <xsd:element name="invite" type="p2pio:InviteType"/>
  <xsd:element name="ok" type="p2pio:OkType"/>
  <xsd:element name="error" type="p2pio:ErrorType"/>
  <xsd:element name="itemSet" type="p2pio:ItemSetType"/>
  <xsd:element name="scope" type="p2pio:ScopeType"/>
  <xsd:element name="stillOpen" type="p2pio:StillOpenType"/>
  <xsd:element name="transactionID" type="xsd:string"/>

  <xsd:complexType name="OpenType">
    <xsd:sequence>
      <xsd:element name="transactionID" type="xsd:string"/>
      <xsd:element name="query" type="p2pio:AnyQueryType"/>
      <xsd:element name="scope" type="p2pio:ScopeType"/>
      <xsd:element name="responseMode" type="p2pio:ResponseModeType
          "/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="AnyQueryType">
    <xsd:sequence>
      <xsd:any processContents="lax">
      </xsd:any>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="ResponseModeType">
```

```
  <xsd:choice>
    <xsd:element name="routedResponseMode">
      <xsd:complexType/>
    </xsd:element>
    <xsd:element name="directResponseMode">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="locator" type="p2pio:LocatorType"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:choice>
</xsd:complexType>

<xsd:complexType name="ReceiveType">
  <xsd:sequence>
    <xsd:element name="transactionID" type="xsd:string"/>
    <xsd:element name="min" type="xsd:positiveInteger"/>
    <xsd:element name="max" type="xsd:positiveInteger"/>
    <xsd:element name="timeout" type="xsd:dateTime" minOccurs="0"
        />
  </xsd:sequence>
  <xsd:attribute name="mode">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="single"/>
        <xsd:enumeration value="multi"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:complexType>

<xsd:complexType name="SendType">
  <xsd:sequence>
    <xsd:element name="transactionID" type="xsd:string"/>
    <xsd:element name="itemSet" type="p2pio:ItemSetType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="FinalSendType">
  <xsd:sequence>
    <xsd:element name="transactionID" type="xsd:string"/>
    <xsd:element name="itemSet" type="p2pio:ItemSetType"/>
    <xsd:element name="isStillOpen" type="p2pio:StillOpenType"
        minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="CloseType">
  <xsd:sequence>
    <xsd:element name="transactionID" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="InviteType">
  <xsd:sequence>
    <xsd:element name="transactionID" type="xsd:string"/>
```

```
    <xsd:element name="locator" type="p2pio:LocatorType"
        minOccurs="0"/>
    <xsd:element name="isStillOpen" type="p2pio:StillOpenType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="OkType">
  <xsd:sequence>
    <xsd:element name="transactionID" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ErrorType">
  <xsd:sequence>
    <xsd:element name="transactionID" type="xsd:string"/>
    <xsd:element name="code" type="p2pio:ErrorCodeType"/>
    <xsd:element name="codeAsText" type="xsd:string"/>
    <xsd:element name="cause" type="xsd:string" minOccurs="0"
        maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ItemSetType">
  <xsd:sequence>
    <xsd:any processContents="lax" minOccurs="0" maxOccurs="
        unbounded">
    </xsd:any>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ExtensionType">
  <xsd:sequence>
    <xsd:any processContents="lax" minOccurs="0" maxOccurs="
        unbounded">
    </xsd:any>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ScopeType">
  <xsd:sequence>
    <xsd:element name="timeout" type="xsd:dateTime"/>
    <xsd:element name="maxHops" type="xsd:unsignedInt"/>
    <xsd:element name="neighborSelection" type="
        p2pio:AnyQueryType" minOccurs="0"/>
    <xsd:element name="extension" type="p2pio:ExtensionType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="LocatorType">
  <xsd:sequence>
    <xsd:element name="endpoint" type="xsd:anyURI" minOccurs="1"
        maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="StillOpenType">
  <xsd:sequence>
```

```
      <xsd:element name="locallyAvailable" type="xsd:unsignedInt"
          minOccurs="0"/>
      <xsd:element name="globallyAvailable" type="xsd:unsignedInt"
          minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:simpleType name="ErrorCodeType">
    <xsd:restriction base="xsd:unsignedInt">
      <xsd:minInclusive value="100"/>
      <xsd:maxInclusive value="999"/>
    </xsd:restriction>
  </xsd:simpleType>

</xsd:schema>
```

References

[1] Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int'l. Journal of Supercomputer Applications*, 15(3), 2001.

[2] Gnutella Community. Gnutella Protocol Specification v0.4. dss.clip2.com/GnutellaProtocol04.pdf.

[3] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[4] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical report, U.C. Berkeley UCB//CSD-01-1141, 2001.

[5] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, 2001.

[6] M. van Steen, P. Homburg, and A. Tanenbaum. A wide-area distributed system. *IEEE Concurrency*, 1999.

[7] Bernard Traversat, Mohamed Abdelaziz, Mike Duigou, Jean-Christophe Hugly, Eric Pouyoul, and Bill Yeager. Project JXTA Virtual Network, 2002. White Paper, http://www.jxta.org.

[8] Steven Waterhouse. JXTA Search: Distributed Search for Distributed Networks, 2001. White Paper, http://www.jxta.org.

[9] Project JXTA. JXTA v1.0 Protocols Specification, 2002. http://spec.jxta.org.

[10] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[11] UDDI Consortium. UDDI: Universal Description, Discovery and Integration. http://www.uddi.org.

[12] Brian Tierney, Ruth Aydt, Dan Gunter, Warren Smith, Valerie Taylor, Rich Wolski, and Martin Swany. A Grid Monitoring Architecture. Technical report, Global Grid Forum Informational Document, January 2002. http://www.gridforum.org.

[13] Jason Lee, Dan Gunter, Martin Stoufer, and Brian Tierney. Monitoring Data Archives for Grid Environments. In *Proc. of the Int'l. IEEE/ACM Supercomputing Conference (SC 2002)*, Baltimore, USA, November 2002. IEEE Computer Society Press.

[14] Ashley Beitz, Mirion Bearman, and Andreas Vogel. Service Location in an Open Distributed Environment. In *Proc. of the Int'l. Workshop on Services in Distributed and Networked Environements*, Whistler, Canada, June 1995.

[15] Object Management Group. Trading Object Service. *OMG RPF5 Submission*, May 1996.

[16] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. *IETF RFC 1777*, March 1995.

[17] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid Information Services for Distributed Resource Sharing. In *Tenth IEEE Int'l. Symposium on High-Performance Distributed Computing (HPDC-10)*, San Francisco, California, August 2001.

[18] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7), July 1999.

[19] Erik Guttman. Service Location Protocol: Automatic Discovery of IP Network Services. *IEEE Internet Computing Journal*, 3(4), 1999.

[20] Weibin Zhao, Henning Schulzrinne, and Erik Guttman. mSLP - Mesh Enhanced Service Location Protocol. In *Proc. of the IEEE Int'l. Conf. on Computer Communications and Networks (ICCCN'00)*, Las Vegas, USA, October 2000.

[21] Steven E. Czerwinski, Ben Y. Zhao, Todd Hodes, Anthony D. Joseph, and Randy Katz. An Architecture for a Secure Service Discovery Service. In *Fifth Annual Int'l. Conf. on Mobile Computing and Networks (MobiCOM '99)*, Seattle, WA, August 1999.

[22] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proc. of the Symposium on Operating Systems Principles*, Kiawah Island, USA, December 1999.

[23] Wolfgang Hoschek. Peer-to-Peer Grid Databases for Web Service Discovery. *Journal of Concurrency and Computation: Practice and Experience (CCPE)*, 2002. Wiley Press.

[24] Wolfgang Hoschek. Dynamic Timeouts and Neighbor Selection Queries in Peer-to-Peer Networks. In *Int'l. Conf. on Networks, Parallel and Distributed Processing and Applications (NPDPA 2002)*, Tsukuba, Japan, October 2002.

[25] M. Mealling, P. Leach, and R. Salz. A UUID URN Namespace. *IETF Internet Draft draft-mealling-uuid-urn-00.txt*, October 2002.

[26] Ron Rivest. The MD5 message-digest algorithm. *IETF RFC 1321*, April 1992.

[27] National Institute of Standards and Technology. Secure Hash Standard. Technical report, FIPS 180-1, Washington, D.C., April 1995.

[28] Wolfgang Hoschek. *A Unified Peer-to-Peer Database Framework for XQueries over Dynamic Distributed Content and its Application for Scalable Service Discovery.* PhD Thesis, Technical University of Vienna, March 2002.

[29] Wolfgang Hoschek. A Unified Peer-to-Peer Database Framework for Scalable Service and Resource Discovery. In *Proc. of the 3rd Int'l. IEEE/ACM Workshop on Grid Computing (Grid'2002)*, Baltimore, USA, November 2002. Springer Verlag.

[30] World Wide Web Consortium. XML Schema Part 0: Primer. *W3C Recommendation*, May 2001.