A Unified Peer-to-Peer Database Protocol

Wolfgang Hoschek

CERN IT Division
European Organization for Nuclear Research
1211 Geneva 23, Switzerland
wolfgang.hoschek@cern.ch

Abstract. In a large distributed system spanning many administrative domains such as a Grid, it is desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. However, in such a database system, the set of information tuples in the universe is partitioned over multiple distributed nodes, for reasons including autonomy, scalability, availability, performance and security. This suggests the use of Peer-to-Peer (P2P) query technology.
In this paper, we develop a messaging, communication and network protocol model for the *Unified Peer-to-Peer Database Framework (UPDF)* and the *hyper registry* proposed in our prior studies. Our so-called *Peer Database Protocol (PDP)* has a number of key properties. It is applicable to any node topology and to multiple P2P response modes. To support loosely coupled autonomous Internet infrastructures, the model is connection-oriented and message-oriented. For efficiency, it is stateful at the protocol level, with a transaction consisting of one or more discrete message exchanges related to the same query. It allows for low latency, pipelining, early and/or partial result set retrieval due to synchronous pull, and result set delivery in one or more variable sized batches. It is efficient, due to asynchronous push with delivery of multiple results per batch. It provides resource consumption and flow control on a per query basis, due to the use of a distinct channel per transaction. It is scalable, due to application multiplexing, which allows for very high query concurrency and very low latency, even in the presence of secure TCP connections. To encourage interoperability and extensibility it is fully based on Internet Engineering Task Force (IETF) standards.

1 Introduction

In a large distributed system spanning administrative domains such as a Grid [1], it is desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. Other examples are a (worldwide) service discovery infrastructure for a multi-national organization, a Peer-to-Peer (P2P) file sharing system, the Domain Name System (DNS), the email infrastructure, a monitoring infrastructure for a large-scale cluster of clusters, or an instant messaging and news service. For example, the

European DataGrid (EDG) [2–4] is a software infrastructure that ties together a massive set of globally distributed organizations and computing resources for data-intensive physics analysis applications, including thousands of network services, tens of thousands of CPUs, WAN Gigabit networking as well as Petabytes of disk and tape storage [5]. An enabling step towards increased Grid software execution flexibility is the *web services* vision [2, 6, 7] of distributed computing where programs are no longer configured with static information. Rather, the promise is that programs are made more flexible and powerful by querying Internet databases (registries) at runtime in order to discover information and network attached third-party building blocks. Services can advertise themselves and related metadata via such databases, enabling the assembly of distributed higher-level components. In support of this vision we have introduced the *Web Service Discovery Architecture (WSDA)* [8] and given motivation and justification [9] for the assertion that realistic ubiquitous service and resource discovery requires a rich general-purpose query language such as XQuery [10] or SQL [11]. Based on WSDA, we introduced the *hyper registry* [12], which is a centralized database (node) for discovery of dynamic distributed content.

However, in an Internet discovery database system, the set of information tuples in the universe is partitioned over multiple distributed nodes (peers), for reasons including autonomy, scalability, availability, performance and security. Consequently, we devised the WSDA based *Unified Peer-to-Peer Database Framework (UPDF)* [13, 2], which is unified in the sense that it allows to express specific applications for a wide range of data types (typed or untyped XML, any MIME type [14]), node topologies (e.g. ring, tree, graph), query languages (e.g. XQuery, SQL), query response modes (e.g. Routed, Direct and Referral Response), neighbor selection policies (in the form of an XQuery), pipelining characteristics, timeout and other scope options. In this framework, an *originator* sends a query to an *agent* node, which evaluates it, and forwards it to select *neighbor nodes*. For reliable loop detection in query routes, a query has an identifier and a certain life time. To each query, an originator attaches a static loop timeout and a different transaction identifier, which is a universally unique identifier (UUID). A node maintains a state table of transaction identifiers and returns an error when a query is received that has already been seen and has not yet timed out. In this paper, we develop a messaging model and network protocol for the UPDF framework and the hyper registry.

The design of a messaging model and network protocol for large distributed systems strongly influences system properties such as scalability, efficiency, interoperability, extensibility, reliability, and, of course, limitations in applicability. For example, the success of many applications depends on how fast they can start producing initial/relevant portions of the query result set rather than how fast the entire result set is produced [15]. This is particularly often the case in distributed systems where many nodes are involved in query processing, each of which may be unresponsive for many reasons. The situation is even more pronounced in systems with loosely coupled autonomous nodes. If the messaging model does not support pipelining, a result set has to be delivered with

long latency in a single large batch, even though the query type might allow for pipelining. The key problems are:

- *What messaging and communication model as well as network protocol uniformly supports P2P database queries for a wide range of database architectures and response models such that the stringent demands of ubiquitous Internet discovery infrastructures in terms of scalability, efficiency, interoperability, extensibility and reliability can be met?*
- *In particular, how can one allow for high concurrency, low latency as well as early and/or partial result set retrieval? How can one encourage resource consumption and flow control on a per query basis?*

In this paper, these problems are addressed by developing a unified messaging, communication and network protocol model, collectively termed *Peer Database Protocol (PDP)*. PDP can be mapped in a straightforward manner to several concrete messaging models and network protocols, one of which we explain in detail. Any client (e.g. an originator or a node) can use PDP to query a node, and to retrieve the corresponding result set. While the use of PDP for communication between nodes is mandatory to achieve interoperability, any arbitrary additional protocol and interface may be used for communication between an originator and a node (e.g. a simple stateless SOAP/HTTP request-response or shared memory protocol). For flexibility and simplicity, and to allow for gatewaying, mediation and protocol translation, the relationship between an originator and a node may take any arbitrary form, and is therefore left unspecified.

This paper is organized as follows. Section 2 proposes an abstract messaging model that employs four request messages (QUERY, RECEIVE, INVITE, CLOSE) and a response message (SEND). The semantics of these messages are specified in detail. The notion of a *transaction* as a sequence of one or more message exchanges between two peers (nodes) is explained. We specify the state transitions related to message handling. Section 3 maps in a straightforward manner the abstract PDP messaging model down to the concrete messaging model of the BEEP [16, 17] application level network protocol framework. The permitted kinds of message exchanges are detailed. Message types and their parameters are mapped to XML representations. Section 4 describes how PDP uses the BEEP communication model to carry a message from one peer to another. We explain how to efficiently map sessions and channels to TCP connections. The BEEP network protocol is used in specifying how to handle asynchrony, encoding, framing, authentication, privacy and reporting. Section 5 compares our approach with related work. Finally, Section 6 summarizes and concludes this paper. We also outline interesting directions for future research.

2 Abstract PDP Messaging Model

The abstract PDP messaging model employs four request messages (QUERY, RECEIVE, INVITE, CLOSE) and a response message (SEND). A *transaction*

is a sequence of one or more message exchanges between two peers (nodes) for a given query. An example transaction is a QUERY-RECEIVE-SEND-RECEIVE-SEND-CLOSE sequence[1]. This non-trivial transaction model is in contrast to a simpler model where a transaction consists of a single request-response exchange (e.g. HTTP). The former model is stateful whereas the latter is stateless. A peer can concurrently handle multiple independent transactions. The messages have the following semantics:

- **QUERY.** A QUERY message is forwarded along node hops through the P2P node topology. The message contains the query itself as well as a transaction identifier. The QUERY message also contains scope hints such as a loop timeout, abort timeout, radius and a neighbor selection query [13, 2]. It may optionally also contain a hint indicating what response mode should be used (Routed Response or Direct Response [13, 2]). Under Direct Response, also the service link or description of the agent node must be included, so that nodes with matches can invite the agent to retrieve the result set. Optionally, the identity of the originator may be included to allow for authorization decisions where applicable. A node accepting a QUERY message returns immediately without any results. Results are explicitly requested via a subsequent RECEIVE message.

- **RECEIVE.** A RECEIVE message is used by a client to request query results from another node. It requests the node to SEND a batch of at least `N` and at most `M` results from the (remainder of the) result set. This corresponds to the `next()` method of an iterator (operator). We have $1 \leq N \leq M$. For example, a low latency use case can use `N=1, M=10` to indicate that at least one and at most ten results should be delivered by the next batch. `N=M=infinity` indicates that all remaining results should be send in a single large batch.
  A client can successively issue multiple RECEIVE messages until the result set is exhausted. A client need not retrieve all results from the entire result set. For example, after the first batch of 10 results it may issue a CLOSE request.
  A RECEIVE request contains a parameter that asks to deliver SEND messages in either synchronous (pull) or asynchronous (push) mode. In synchronous mode a single RECEIVE request must precede every single SEND response. An example sequence is RECEIVE-SEND-RECEIVE-SEND. In asynchronous mode a single RECEIVE request asks for a sequence of successive SEND responses. A client need not explicitly request more results, as they are automatically pushed in a sequence of zero or more SENDs. An example sequence is RECEIVE-SEND-SEND-SEND.

- **SEND.** When a node accepts a RECEIVE message, it responds with a SEND message, containing a batch with `P` results from the (remainder of

[1] This notion is entirely unrelated to the notion used in database systems where a transaction is an atomic unit of database access which is either completely executed or not executed at all [18].

the) result set. We have `P` $\leq$ `M`. We may, but need not, have `N` $\leq$ `P`. For example, less than `N` results may be delivered when the entire query result set is exhausted, or if the node decides to override and decrease `N` for reasons including resource consumption control.
A SEND message also contains the number `R` of remaining results currently available for immediate non-blocking delivery with the next SENDs (`nonBlockingResultsAvailable`). Usually `R` is greater than zero. `R=0` can indicate that remote nodes have not yet delivered results necessary to return more than zero results with the next SEND. `R=-1` indicates that the batch contains the last results as the result set is exhausted. No more RECEIVE messages must be issued after that point. `R=-2` indicates that the number is unknown.
A SEND message also contains the current estimate `Q` of the remaining total result set size, irrespective of blocking (`estimatedResultsAvailable`). The actual number of results that can (later) be delivered may be larger. It should not be smaller, except if other nodes fail to deliver their suggested results. Usually `Q` is greater or equal to zero. `R=-1` implies `Q=-1`, indicating that the result set is definitely exhausted. `Q=-2` indicates that the number is unknown.

- **CLOSE.** A client may issue a CLOSE message to inform a node that the remaining results (if any) are no longer needed and can safely be discarded. A CLOSE message responds immediately with an acknowledgement. At the same time, the node asynchronously forwards the CLOSE to neighbors involved in result set delivery, which in turn forward the CLOSE to their neighbors, and so on. Being informed of a CLOSE allows a node to release resources as early as possible. Strictly speaking, a client need not issue a CLOSE, and a node need not forward further a CLOSE, because a query eventually times out anyway. Even though this is considered misbehavior, a node must continue to operate reliably under such conditions.

- **INVITE.** INVITE messages only apply to Direct Response mode [2, 13]. A node forwards the query to the nodes obtained from neighbor selection without ever waiting for their result sets. The node only applies the query to its local database. If the local result set is not empty, the node 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 a synchronous or asynchronous manner (see above). An INVITE message also contains the number `R` of results currently available for immediate non-blocking delivery (`nonBlockingResultsAvailable`). `R` must be greater than or equal to zero. The message also contains the current estimate `Q` of the remaining total result set size, irrespective of blocking (`estimatedResultsAvailable`). `Q` must be greater than zero.

A peer may periodically discover other peers and announce its presence. Peer discovery uses a QUERY that selects all tuples with peer service descriptions. For presence announcement, a peer additionally includes its service description as optional QUERY data. Explicit PING/PONG messages [19] are unnecessary.

Clearly a RECEIVE request may cause cascading RECEIVEs through the nodes of the P2P topology, followed by cascading SEND responses backwards. In the worst case every RECEIVE cascades through a large number of node hops, incurring prohibitive latencies. This highlights the importance of (appropriately sized) batched delivery, which greatly reduces the number of hops incurred by a single RECEIVE. Also, note that the I/O of a node need not be driven strictly by client demand. For example, in an attempt to reduce latency, a node accepting a QUERY may already prefetch query results from its neighbors even though it has not yet seen the corresponding RECEIVE request from its client.

State Transitions. A node maintains a state table. For each query at least the transaction identifier, abort timeout, loop timeout and an open/closed state flag are kept. An example state table reads as follows:

Transaction Identifier	**Abort Timeout**	**Loop Timeout**	**State**
100	20	30	Closed
200	50	60	Open

A query is *known* to a node if the state table already holds a transaction identifier equal to the transaction identifier of the query. Otherwise, it is said to be *unknown*. A known query can be in two states: *open* or *closed*. Let us discuss the state transitions from *unknown* to *open* to *closed* and back to *unknown* state (also summarized in Figure 1).

- **Open.** When an *unknown* query arrives with a QUERY message, it moves into *open* state. When a query moves into *open* state, it becomes known and is forwarded to the neighbors obtained from neighbor selection.

- **Closed.** A query moves from *open* into *closed* state when its abort timeout has been reached, or if the result set is exhausted by the final SEND, or if a client issues a CLOSE to indicate that it is no longer interested in the (remainder of the) result set, or if one of several errors occur. Under direct response, a query to a non-agent node *also* moves from *open* into *closed* state if the query produces no local results, or if it does produce local query results but the INVITE request is not accepted by the agent.
  In any case, when a query moves into *closed* state, a CLOSE request is asynchronously forwarded to all dependents in order to inform them as well. A node depends on a set of other nodes (*dependents*) that are involved in result set delivery. Under Routed Response, the dependants are the nodes obtained from neighbor selection. Under Direct Response, the dependents of an agent are the nodes from which the agent has accepted an INVITE message, whereas all other nodes have no dependents.

- **Unknown.** A query moves from *closed* state into *unknown* state when its loop timeout has been reached. In other words, the query is deleted from the state table.

– **Message Acceptance and Rejection.** A QUERY request is accepted if the query is *unknown*. If an already known QUERY arrives, this usually indicates loop detection [2, 13]. The message is rejected with an error (e.g. "transaction identifier already in use"). When a message other than QUERY arrives that has an unknown transaction identifier, it is rejected with an error (e.g. "transaction identifier unknown"). RECEIVE, SEND, CLOSE and INVITE messages are accepted for a query in *open* state. No message for a query in *closed* state is accepted; the response to a message is always an error (e.g. "transaction identifier already closed").

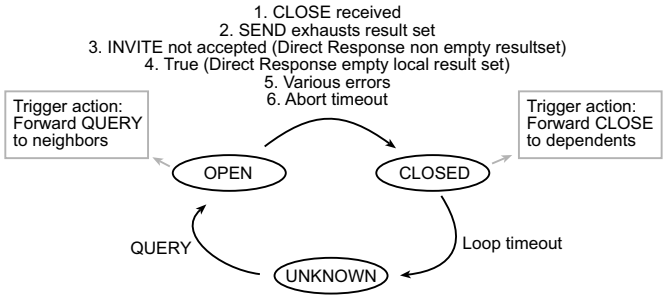

Fig. 1. Node State Transitions.

3 Concrete Messaging Model

BEEP Messaging Model. The BEEP application level network protocol framework [16, 17] is an IETF standard designed for connection-oriented (ordered, reliable, congestion sensitive), message-oriented (loosely coupled, structured data), asynchronous (peer-to-peer, allowing client-server) communications. The messaging model of the framework defines one *request* message class (MSG) and four *response* messages classes (RPY, ERR, ANS, NULL). 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 response to a MSG message may be an *error* (ERR), a *reply* (RPY) or a sequence of zero or more *answers* (ANS), followed by a *null terminator* message (NULL). The exchange patterns are summarized as follows:

```
MSG --> RPY | (ANS [0..N], NULL) | ERR
```

PDP Messaging Model. In a straightforward manner, the abstract PDP messaging model is now mapped down to the concrete BEEP messaging model. The

BEEP framework explicitly expects each message class to be extended by applications as necessary. Accordingly, the messages QUERY, RECEIVE, SEND, CLOSE, INVITE are refined, yielding three request MSG types (MSG QUERY, MSG RECEIVE, MSG INVITE), two reply message types (RPY SEND, RPY OK), one answer message type (ANS SEND), and the ERR error type. The RPY OK and ERR message type are introduced because any realistic messaging model must deal with acknowledgments and errors. The following message exchanges are permitted:

```
MSG QUERY   --> RPY OK | ERR
MSG RECEIVE --> RPY SEND | (ANS SEND [0:N], NULL) | ERR
MSG INVITE  --> RPY OK | ERR
MSG CLOSE   --> RPY OK | ERR
```

The messaging model is exemplified by the following corresponding message flows from client to server ("`-->`") and back ("`<--`"):

Routed Synchr. Response	**Routed Asynchr. Response**	**Direct Synchr. Response**	**Direct Asynchr. Response**
`--> MSG QUERY`	`--> MSG QUERY`	`--> MSG QUERY`	`--> MSG QUERY`
`<-- RPY OK`	`<-- RPY OK`	`<-- RPY OK`	`<-- RPY OK`
`--> MSG RECEIVE`	`--> MSG RECEIVE`		
`<-- RPY SEND`	`<-- ANS SEND`	`<-- MSG INVITE`	`<-- MSG INVITE`
`--> MSG RECEIVE`	`<-- ANS SEND`	`--> RPY OK`	`--> RPY OK`
`<-- RPY SEND`	`<-- NULL`	`--> MSG RECEIVE`	`--> MSG RECEIVE`
`--> MSG CLOSE`	`--> MSG CLOSE`	`<-- RPY SEND`	`<-- ANS SEND`
`<-- RPY OK`	`<-- RPY OK`	`--> MSG RECEIVE`	`<-- ANS SEND`
		`<-- RPY SEND`	`<-- NULL`
		`--> MSG CLOSE`	`--> MSG CLOSE`
		`<-- RPY OK`	`<-- RPY OK`

Concrete PDP Message Representations. Message types and their parameters can be mapped to multiple representations. For simplicity and flexibility, PDP uses straightforward XML [20] representations, as depicted in Figure 2. Without loss of generality, example query expressions (e.g. user query, merge query and neighbor selection query) are given in the XQuery language [10], as detailed in [12]. Other query languages such as XPath, SQL [11] or LDAP [21] or subscription interest statements could also be used. Indeed, the messages and network interactions required to support publish-subscribe or event trigger systems do not differ at all from the ones presented in this paper.

4 Communication Model and Network Protocol

BEEP Communication Model. In this subsection, the communications between two peers (e.g. originator and agent node, or node and another node) are de-

```
<MSG_QUERY transactionID = "12345">
  <query>
        <userquery> RETURN /tupleset/tuple </userquery>
        <mergequery unionizer="UNION"> RETURN /tupleset/tuple </mergequery>
  </query>
  <scope loopTimeout = "2000000000000" abortTimeout = "1000000000000"
         logicalRadius = "7" physicalRadius = "4"
         maxResults = "100"  maxResultsBytes = "100000">
         <neighborSelectionQuery>         <!-- implements broadcasting -->
            RETURN /tupleset/tuple[@type="service"
              AND content/service/interface[@type="Consumer-1.0"]
              AND content/service/interface[@type="XQuery-1.0"]]
         </neighborSelectionQuery>
  </scope>
  <options>
         <responseMode> routed </responseMode>
         <originator> fred@example.com </originator>
  </options>
</MSG_QUERY>

<MSG_RECEIVE transactionID = "12345">
  <mode minResults = "1" maxResults = "10">  synchronous  </mode>
</MSG_RECEIVE>

<RPY_SEND transactionID = "12345">
  <data nonBlockingResultsAvailable = "-1" estimatedResultsAvailable = "-1">
    <tupleset TS4="100">
       <tuple link="http://sched.infn.it:8080/pub/getServiceDescription"
              type="service" ctx="child" TS1="20" TC="25" TS2="30" TS3="40">
          <content>
             <service> service description B goes here </service>
          </content>
       </tuple>
       ... more tuples can go here ...
    </tupleset>
  </data>
</RPY_SEND>

<ANS_SEND transactionID = "12345">
    structure is identical to RPY_SEND (see above) ...
</ANS_SEND>

<MSG_INVITE transactionID = "12345">
  <avail nonBlockingResultsAvailable="50" estimatedResultsAvailable="100"/>
</MSG_INVITE>

<MSG_CLOSE transactionID = "12345" code="555"> maximum idle time exceeded
</MSG_CLOSE>

<RPY_OK transactionID = "12345"/>

<ERR transactionID = "12345" code="550"> transaction identifier unknown </ERR>
```

Fig. 2. Concrete Messages.

scribed by the BEEP communication model, which well fits the requirements of our (non-trivial) messaging model. The model has the following properties:

- **Session, Channel, Message and Frame.** Two peers establish a *session* for communication. Within a session, one or more concurrent *channels* can be established. A channel carries zero or more *messages*. A message can have arbitrary length and content. A message is segmented into one or more *frames* of variable length. A session is established by an initiator for communication with a listener. Within a session, the peer that awaits new channels is acting in the server role, and the other peer, which establishes a channel to the server, is acting in the client role. In P2P style, both initiator and listener may (but need not) act as client and server at the same time.

- **Intra-channel.** Within a channel, all messages are processed in serial order. The server must generate responses in the same order as corresponding request messages are received. One or more request messages may be issued without waiting to receive the corresponding responses. That is, a channel provides *pipelining*. To this end, each request message carries an integer identifier that is unique within the channel. Responses to the message carry the same identifier.

- **Inter-channel.** Channels are isolated from each other, and therefore handle asynchrony and multiplexing. A channel cannot "see" or interfere with messages from other channels. There are no constraints on the processing order for different channels. In other words, inter-channel messages may be unordered.

- **Flow control.** In all likelihood, the concurrent channels of a session are carried over the same physical network cable. Consequently, flow control policy issues arise: If more than one channel offers a frame to send, which frame should be chosen? What is a good size for a frame? A peer may implement any policy it sees fit. For example, it may attempt to prevent starvation and encourage fairness. A slow channel should not be able to monopolize the session. Large messages may be segmented into multiple frames, and different channels may be served in round-robin fashion or according to priorities.

PDP Communication Model. We now describe how PDP uses the BEEP communication model for its purposes. Because of the non-trivial PDP transaction model, we use *one channel per distinct transaction* to isolate communication referring to different queries and to ensure that messages of a transaction are processed in serial order. This also provides for resource consumption and flow control on a per query basis. The concepts of session and channel can be mapped to concrete physical entities in several ways:

- **Session.** There are two options: one session per originator, or one session per initiator (neighbor node). Under the former option, a new session between two peers is established whenever a query from a previously unknown

originator is accepted. The session is shared by all queries from the given originator. This option does not scale well in the presence of many concurrent originators. In contrast, under the latter option, a new session is established whenever a new node publishes itself as neighbor (more lazy: when the first query exchange with a neighbor happens). Irrespective of how many originators issue queries, the session persists until a node leaves the network. Since neighbors do not join and leave very frequently, this option involves less latency because session establishment occurs less often.

– **Channel.** As has been noted, there exists one channel per transaction (query). There are two options to map TCP connections: one TCP connection per channel (*TCP multiplexing*, *TM*) and one TCP connection per session (*application multiplexing, AM*).
TM is easy to implement because multiplexing is directly supported by the TCP stack, which natively handles multiple concurrent TCP connections. An application need not bother how to implement multiplexing and flow control, but it also has few means to control it. For simplicity, almost all network protocols use TM. Under AM, all channels of a session share a single TCP connection. Typically, each channel has an associated memory buffer. AM is much more complex because multiplexing must be supported on top of the TCP stack, at the application level. Note, however, that the complexities involved are well taken care of by existing commodity software frameworks such as `beepcore` [22]. TM has the distinct disadvantage of being much less efficient in the presence of high frequency channel creation. With new queries (channels) arriving at high frequency, TM encounters serious latency limitations due to the very expensive nature of secure (and even insecure) TCP connection setup. Even if TCP connections are kept alive, pooled and reused, at least `N*neighbors` TCP connections are needed under broadcast to handle `N` concurrent queries. While this solution may be perfectly adequate for small special-purpose networks, it clearly does not scale well to the stringent demands of a ubiquitous Internet infrastructure such as service discovery. This is precisely the demanding scenario AM is designed for: Channel establishment only requires a single message exchange over an already existing TCP connection. If channels are pooled and reused, channel establishment is a null operation and does not involve any network communication.

It appears unnecessary and inefficient to setup up a new session for each originator. Hence, PDP uses one session per initiator (neighbor node). For simplicity and easy authorization, PDP actually uses two sessions per initiator (neighbor node). One session is used for traffic related to incoming queries (queries posed to the node), the other for traffic related to outgoing queries (queries the node poses to another node). A node with `N` neighbors has `N` incoming and `N` outgoing sessions. In a successful P2P network, queries do indeed arrive at high frequencies. Session establishment may be heavyweight, but channel establishment must be lightweight. Hence, application multiplexing is chosen. A node with `N` neighbors has `N` incoming and `N` outgoing TCP connections.

BEEP Network Protocol. The BEEP network protoccol uses channels for asynchrony (handling independent exchanges). Its transport mapping to TCP uses application multiplexing (one TCP connection per session) with sliding windows [17]. Each channel has a sliding window that indicates the number of payload octets that a peer may transmit before receiving further permission to transmit. MIME [14] with a default of `text/xml` is used for encoding (representing messages). Octet counting with trailers is used for framing (delimiting messages). SASL [23] and/or TLS/SSL [24] are used for authentication (verifying user identities) and privacy (protecting against third-party interception). 3-digit and localized textual diagnostics are used for reporting (conveying status information such as errors). We note that BEEP can be mapped to any reliable transport layer (TCP is merely the default).

PDP Network Protocol. Any network protocol must deal with a set of common problems. The BEEP framework was introduced to avoid the need to reinvent solutions to common problems. We propose to adopt the framework because it integrates existing best-of-breed standards. PDP encodes message types and their parameters with the straightforward XML representations given in Section 3, using the MIME type `text/xml`, which is the default in BEEP. For Grid applications, PDP uses TLS/SSL within the context of the Grid Security Infrastructure (GSI) [25].

5 Related Work

RDBMS. The network protocols of Relational Database Management Systems are designed with a tight focus on a single node architecture and response model. For maximum efficiency, communication is tightly coupled, for example with low overhead Inter Process Communication (IPC) mechanisms carried over more efficient layers than TCP. Like our approach, RDBMS protocols also are stateful and allow for low latency, pipelining, early and/or partial result set retrieval due to synchronous pull, and result set delivery in one or more variable sized batches. They provide very strong functionality to provide for resource consumption and flow control on a per query and/or per user basis. Low-level RDBMS interfaces such as Oracle's OCI [26] allow for application multiplexing. High-level access APIs such as JDBC [27] do not provide access to such facilities; they use less scalable TCP connection pooling instead. RDBMS protocols are closed and proprietary (except for open source products), and hence unsuitable for Internet-level interoperability and extensibility.

LDAP and MDS. The Lightweight Directory Access Protocol (LDAP) [21] allows for multi-level hierarchical topologies as well as normal and referral response modes. It does not support arbitrary topologies and Direct Response mode. The Metacomputing Directory Service (MDS) [28, 29] additionally supports routed response mode but otherwise has the same properties as LDAP. Like our approach, both protocols are stateful as well as connection and message-oriented.

They do not support synchronous pull, and result set delivery in one or more variable sized batches. Synchronous paging behavior has been proposed [30], but this is still inefficient, because each response message still contains a single entry only. LDAP and MDS do support asynchronous push. They do not provide for resource consumption and flow control on a per query basis. They lack a concept that concentrates and serializes all messages related to a query, like a BEEP channel. LDAP has a notion of application multiplexing that is not equivalent to ours. The fact that messages may be unordered is dictated by the LDAP network protocol. The BEEP network protocol guarantees for ordered message delivery. If LDAP were used for PDP messaging, the parameters `Q` and `R` of a SEND message would be meaningless. Likewise, the server response for a CLOSE request would be allowed to "overtake" the SEND responses for prior RECEIVE requests, which violates pipelining semantics. Like BEEP, LDAP is an IETF standard.

Gnutella and Freenet. Gnutella [19] and Freenet [31] support queries in arbitrary graph topologies but only a single response mode. Like our approach, their protocols are stateful as well as connection and message-oriented. They do not support synchronous pull but they do support asynchronous push with one or more variable sized batches. Like LDAP and MDS, they do not provide for resource consumption and flow control on a per query basis. However, they do have a notion of application multiplexing that is equivalent to ours for the purpose of result *set* retrieval. Their protocol specifications are not closed and proprietary, but they are ad-hoc specifications without any relation to an open IETF standard and its implied quality in terms of interoperability and extensibility.

6 Conclusions

In this paper, we develop a messaging, communication and network protocol model, collectively termed *Peer Database Protocol (PDP)*. PDP supports P2P database queries for a wide range of database architectures and response models such that the stringent demands of ubiquitous Internet infrastructures in terms of scalability, efficiency, interoperability, extensibility and reliability can be met.

PDP has a number of key properties. It is applicable to any node topology (e.g. star, ring, tree, graph) and to multiple P2P response modes. To support loosely coupled autonomous Internet infrastructures, the model is connection-oriented (ordered, reliable, congestion sensitive) and message-oriented (loosely coupled, operating on structured data). For efficiency, it is stateful at the protocol level, with a transaction consisting of one or more discrete message exchanges related to the same query. It allows for low latency, pipelining, early and/or partial result set retrieval due to synchronous pull, and result set delivery in one or more variable sized batches. It is efficient, due to asynchronous push with delivery of multiple results per batch. It provides for resource consumption and flow control on a per query basis, due to the use of a distinct channel per transaction. It is scalable, due to application multiplexing, which allows for very

high query concurrency and very low latency, even in the presence of secure TCP connections. To encourage interoperability and extensibility it is fully based on Internet Engineering Task Force (IETF) standards.

These key properties distinguish our approach from related work, which individually addresses some, but not all of the above issues. We are not aware of related work that proposes a uniform messaging model that is applicable to any node topology and at the same time to multiple P2P response modes. Some related work does not apply to loosely coupled autonomous database nodes (RDBMS). Some protocols are not stateful at the protocol level (HTTP based mechanisms). Some do not support synchronous pull (LDAP, MDS, Gnutella, Freenet) and result set delivery in one or more variable sized batches (LDAP, MDS, HTTP based mechanisms). Some do not support asynchronous push with delivery of multiple results per batch (LDAP, MDS, HTTP based mechanisms). Some do not provide for resource consumption and flow control on a per query basis (LDAP, MDS, Gnutella, Freenet, HTTP based mechanisms). Some lack application multiplexing for scalable query concurrency (some RDBMS drivers, HTTP based mechanisms, LDAP, MDS). Some do not encourage interoperability and extensibility based on open IETF standards (RDBMS, Gnutella, Freenet).

Interesting directions for future research include investigating the use of SOAP [32] as a high-level tool for PDP messaging. Most commonly, HTTP 1.1 [33] is used as SOAP transport. However, SOAP is transport protocol independent. For strongly increased efficiency and low latency, SOAP should be carried over BEEP. See [34] for an IETF draft specifying a straightforward SOAP/BEEP binding.

Acknowledgements. We would like to thank Erwin Laure for thoroughly reviewing this paper and for contributing detailed suggestions for structural improvements.

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. 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.
3. Ben Segal. Grid Computing: The European Data Grid Project. In *IEEE Nuclear Science Symposium and Medical Imaging Conference*, Lyon, France, October 2000.
4. Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid Project. In *1st IEEE/ACM Int'l. Workshop on Grid Computing (Grid'2000)*, Bangalore, India, December 2000.
5. Large Hadron Collider Committee. Report of the LHC Computing Review. Technical report, CERN/LHCC/2001-004, April 2001. http://lhc-computing-review-public.web.cern.ch/lhc-computing-review-public/Public/Report_final.PDF.

6. Ian Foster, Carl Kesselman, Jeffrey Nick, and Steve Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, January 2002. http://www.globus.org.
7. P. Cauldwell, R. Chawla, Vivek Chopra, Gary Damschen, Chris Dix, Tony Hong, Francis Norton, Uche Ogbuji, Glenn Olander, Mark A. Richman, Kristy Saunders, and Zoran Zaev. *Professional XML Web Services*. Wrox Press, 2001.
8. Wolfgang Hoschek. The Web Service Discovery Architecture. In *Proc. of the Int'l. IEEE/ACM Supercomputing Conference (SC 2002)*, Baltimore, USA, November 2002. IEEE Computer Society Press.
9. Wolfgang Hoschek. A Data Model and Query Language for Service Discovery. Technical report, DataGrid-02-TED-0409, April 2002.
10. World Wide Web Consortium. XQuery 1.0: An XML Query Language. *W3C Working Draft*, December 2001.
11. International Organization for Standardization (ISO). Information Technology-Database Language SQL. *Standard No. ISO/IEC 9075:1999*, 1999.
12. Wolfgang Hoschek. A Database for Dynamic Distributed Content and its Application for Service and Resource Discovery. In *Int'l. IEEE Symposium on Parallel and Distributed Computing (ISPDC 2002)*, Iasi, Romania, July 2002.
13. Wolfgang Hoschek. A Unified Peer-to-Peer Database Framework and its Application for Scalable Service Discovery. In *Proc. of the 3rd Int'l. IEEE/ACM Workshop on Grid Computing (Grid'2002)*, Baltimore, USA, November 2002. Springer Verlag.
14. N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. *IETF RFC 2045*, November 1996.
15. T. Urhan and M. Franklin. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. *The Very Large Database (VLDB) Journal*, 2001.
16. Marshall Rose. The Blocks Extensible Exchange Protocol Core. *IETF RFC 3080*, March 2001.
17. Marshall Rose. Mapping the BEEP Core onto TCP. *IETF RFC 3081*, March 2001.
18. Stefano Ceri and Giuseppe Pelagatti. *Distributed Databases - Principles and Systems*. McGraw-Hill Computer Science Series, 1985.
19. Gnutella Community. Gnutella Protocol Specification v0.4. dss.clip2.com/GnutellaProtocol04.pdf.
20. World Wide Web Consortium. Extensible Markup Language (XML) 1.0. *W3C Recommendation*, October 2000.
21. W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. *IETF RFC 1777*, March 1995.
22. Beepcore Community. The beepcore open source project for Java, C and Tcl. http://www.beepcore.org.
23. J. Myers. Simple Authentication and Security Layer (SASL). *IETF RFC 2222*, October 1997.
24. T. Dierks and C. Allen. The TLS Protocol Version 1.0. *IETF RFC 2246*, January 1999.
25. R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, and V. Welch. A National-Scale Authentication Infrastructure. *IEEE Computer*, 33(12), 2000.
26. Oracle Corp. Oracle Call Interface Programmer's Guide, January 2002. Release 9.0.1, Part Number A89857-01.
27. Donald Bales. *Java Programming with Oracle JDBC*. O'Reilly, December 2001. ISBN 0-596-00088-x.

28. 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.
29. Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith, and Steven Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *6th Int'l. Symposium on High Performance Distributed Computing (HPDC '97)*, 1997.
30. C. Weider, A. Herron, A. Anantha, and T. Howes. LDAP Control Extension for Simple Paged Results Manipulation. *IETF RFC 2696*.
31. 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.
32. World Wide Web Consortium. Simple Object Access Protocol (SOAP) 1.1. *W3C Note 8*, 2000.
33. R. Fielding, J. Gettys, J.C. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. *IETF RFC 2616*. UC Irvine, Digital Equipment Corporation, MIT.
34. E. O'Tuathail and M. Rose. Using the Simple Object Access Protocol (SOAP) in Blocks Extensible Exchange Protocol (BEEP). *IETF RFC 3288*, June 2002.