

Plasma: a graph based distributed computing model

Jeff Rose and Antonio Carzaniga

University of Lugano
Faculty of Informatics
6900 Lugano, Switzerland
{jeffrey.rose,antonio.carzaniga}@lu.unisi.ch

ABSTRACT

We present the concept and design of Plasma, a graph based abstraction and associated framework for distributed computing that is amenable to the development of social network infrastructures. Each peer in a Plasma system maintains a local graph structure which represents its view of the network, as well as the application resources it contains. Distributed algorithms are implemented by issuing graph queries, and the underlying query processor automatically crosses network boundaries, performing the parallel lookup and/or graph transformation. Our graph based abstraction allows for the concise expression of many network algorithms as well as many graph oriented queries which are typical in modeling peer-to-peer and social-network systems. Furthermore, by separating the graph data model and operators from their storage and execution, the Plasma system can use efficient algorithms and optimization techniques to utilize the underlying resources as efficiently as possible. We study the implementation of some representative distributed services in the context of Plasma, finding that it allows for concise implementations, and is amenable to a wide range of uses.

1. INTRODUCTION

Social networks represent a stepping stone in the ongoing process of using the Internet to enable the social manipulation of information and culture. Most current social network sites are implemented as large distributed systems running in centrally controlled data centers; however, the trend in these massively scalable systems is toward the use of peer-to-peer style techniques that offer best-effort services rather than the guaranteed consistency offered by traditional databases and distributed systems. We believe that this trend toward greater distribution will inevitably continue as social networks themselves become federated across a diverse number of sites, and eventually as they take the form of fully distributed peer-to-peer networks. Finding abstractions to simplify the job of working on top of such complexity is the goal of this research.

Our main intuition is to develop a system capable of modeling and supporting both the structural elements

of a network (e.g., peers and connections) and the information that the network maintains (e.g., resources and their relationships, or users and their social connections). In particular, we propose to base our system on a *graph model*. Graphs are one of the most fundamental mathematical abstractions for modeling networks and data; however, it is surprising that this natural abstraction has not yet found its way into the APIs of network systems themselves. In this paper we present initial work on Plasma, a graph abstraction and its associated framework which is designed specifically for distributed systems using information modeled as graphs. The goal of this software abstraction is to allow data and networks to be represented and managed uniformly as data items in a unique sort of graph database, and algorithms to be expressed as access queries on this database.

Peers in a Plasma network maintain a local graph structure representing their view of the distributed system and the data which it contains. Graph queries are used to traverse these structures, and when nodes in the graph which reside on remote peers are encountered, the query processor will transparently issue a sub-query to the remote peer. In this manner, distributed algorithms can be expressed concisely at the level of their graph representation. Below the graph layer, the Plasma graph management system can use efficient algorithms and optimization steps to utilize the underlying resources as efficiently as possible. We motivate the need for this layer of abstraction by a discussion of the social network landscape.

Current application support for social networks is limited to accessing the social graph, but what is necessary is a common abstraction that can be used by applications to take advantage of each others data without imposing new requirements on the overall interface of the system. This requires a data model that is flexible and dynamic so that applications and users are free to add new data and new relations at will.

The primary contribution of this work is the development of a graph abstraction for distributed computing with large scale systems. Plasma introduces a separa-

tion of the logical operations performed over a graph data model, and the underlying network protocol and graph algorithm implementations. In essence, this work attempts to do to large-scale distributed computing and networking what the introduction of the relational model did to data management and databases.

In Section 2 we frame the Plasma system within the context of previous work. In Section 3 we present a formal description of the data model, and then in Section 4 we discuss the query language which is needed to manipulate it. Section 4.1 demonstrates how distributed services are implemented using the graph system, and Section 5 presents the initial Plasma architecture. Finally, Section 7 outlines the future work and concludes.

2. RELATED WORK

The core concept of Plasma, using a graph as the fundamental abstraction for navigating information structures, dates back to some of the first databases. In this paper we revisit the network model [11], and we show that in some sense Charles Bachman’s Turing award vision of “The Programmer as Navigator” [2] should in fact come to fruition.

The network model allowed users to navigate from record to record, but the simple data models of these early systems were closely tied to their physical implementation and lacked expressive operators. Around the same time Codd invented the relational model [4] based on storing data in tables, which introduced a clean logical abstraction that was separated from the details of the physical implementation. The associated relational algebra and logic made it easier to develop database designs, and the focus shifted towards modeling data as seen by the applications and users rather than by the underlying implementation [8]. Although an important evolution in the development of databases, modern applications are running into the limitations of the relational model. The schema is fixed and extending relational databases is a difficult task even in a centralized environment. In a distributed system where peers might be autonomously generating their own structure, the relational model is just not an option.

Semi-structured database systems [7] are focused on storing data items, and typically the relations between this data is treated as a second class feature of the system. Semi-structured databases have been working on the storage of irregular, implicit and partial data where the schema is not restrictive, but as of yet these systems have not been targeted at creating distributed computing platforms. Typically these databases are also tree structured, and although cycles are sometimes possible, the types of operations and queries do not support general graphs.

Unlike the use of RDF graphs in the semantic web, where logical inference engines are expected to operate

over knowledge bases, the goal of Plasma is to serve as a more generic graph modeling system with strong support for distributed computing. In Plasma we do not attempt to model anything in terms of semantics so there is no usage of ontologies, but much of the work related to RDF query languages is still very relevant. In later examples we make use of the SerQL [3] query syntax developed for RDF graphs, and we expect that further insights will come from lessons learned in the semantic web community.

Besides RDF stores, a number of object oriented and graph databases have been proposed in the literature. A recent survey [1] presents a good overview of graph database models. We are not aware of any graph databases that were designed to serve as the basis for distributed computing.

Peer-to-peer (P2P) networking has been working primarily on the scalability issues inherent in distributing resources over a large number of networked processes. There is a dichotomy between the structured and unstructured worlds, while it seems clear that both will be needed. P2P work is still done very close to the metal, so creating abstractions that allow research to be reused is vital for the further development of the area.

Agent systems [9] have been looking at mobile code and the security issues related to distributed processing. Much of this work will be relevant in the future, but for the current work we try to limit this because the goal is to run over heterogeneous systems and the practical implementation and security issues are too great in mobile agent stuff.

3. DATA MODEL

The Plasma data model describes the conceptual tools for representing information in graph structures and the collection of operators that are available to operate on these graphs.

A Plasma graph G , is a set of vertices (nodes) V connected by edges E . Thus $G = (V, E)$. In order to avoid confusion, we limit the use of the term “node” in this paper to meaning a vertex in a graph data structure, and instead use the term “peer” to refer to a process in a distributed system. Similar to graph-based database models such as the Object Exchange Model (OEM) [10], information is stored in nodes that are connected by labeled, directed edges. A node v is a terminal point or an intersection point of a graph, and it is the abstraction of an entity such as a network peer or a person in a social network. Each node in a Plasma graph holds a universally unique, 128-bit identifier (UUID), a set of incoming edges, a set of outgoing edges, and a value. The value can be any type of object, such as an integer, a string, a document, a function or a program. Graphs are rooted purely for convenience so that path expressions can easily start from a known location without

first looking up a node.

An edge e is a link connecting two nodes, where the edge (i, j) has the source node i and the target node j . A link is the abstraction for relationships between nodes such as network connections between peers or personal relationships in a social network. Implemented as a node itself, an edge holds a UUID, a label as its value, and a reference to both the source and target nodes which it connects. As in RDF, this allows edges themselves to have additional metadata attached, such as an edge weight or other contextual information about the relationship. Two nodes in a Plasma graph can be connected by multiple edges in both directions, but they may not have two edges with the same label. The graph database community has explored a number of data model designs [1] which support additional features, but our goal in Plasma is to keep the data model as simple and uniform as possible while not limiting its modeling power.

The structure of information stored in a Plasma graph is not enforced by schemas or ontologies, which allows for the free-form modeling and evolution of data over time. Specific algorithms, libraries, applications and most likely communities will benefit from using shared, agreed-upon structures however, so we believe many informal schemas will arise as usage dictates them.

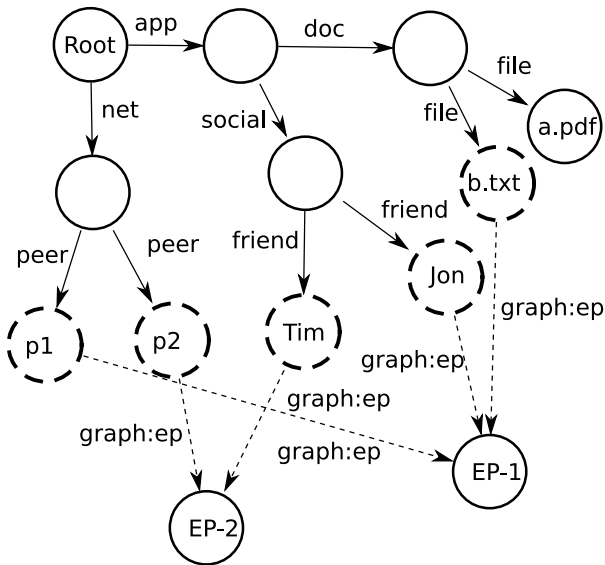


Figure 1: This is an example graph structure that would reside on a single peer in a Plasma network. In many of the example queries it is assumed that the other peers in the network are maintaining graphs corresponding to this structure as well.

An important feature of Plasma graphs is that it supports the concept of *remote nodes*. A remote node is identical to a regular, local node, except that it main-

tains an extra edge connecting to another node that represents the peer endpoint for the graph on which it resides. Figure 1 shows an example graph with two remote nodes representing friends in a social network. This allows the query engine take into account the network location of a node while following a path expression. It also makes it easy to support local caching and proxying of remote nodes. Another feature shown in the example is the use of an edge namespace. By labeling the edges using a prefix separated by a colon, it allows different classes of metadata about nodes to share the same graph without causing confusion during queries and graph navigation. Most users of Plasma will use the default namespace without any prefix, but operational meta-data will typically use them so as not to conflict with user data.

In order to support asynchronous events, a user can attach handler nodes to other nodes in a Plasma graph. These are callable function objects that will be executed when a query either reads, updates or deletes a node, or when it adds or removes an edge to a node. These event callbacks are connected with edges labeled within the *event:* namespace, and they allow for the development of more complex network protocols and applications which want to react to external events, such as a network peer failure or the addition of a new document in a collection.

4. QUERY LANGUAGE

In this section we introduce the features we think are most important in the design of the Plasma query language. Plasma is similar to many graph and semi-structured databases with respect to the basic data model, but we find that existing query languages are not designed with the same goals in mind as Plasma, which is being design to support widely distributed and heterogeneous graph structures. Currently we adopt the basic path syntax of the SeRQL [3] query language for our examples, but the semantics of the languages are quite different.

The basic building block of most graph query languages is the path expression. However, rather than searching a graph database for a matching path, a query in Plasma functions more as navigational constructs that starts at a root node. (The root node defaults to the graph root but can be specified on a per-query basis.) Following is an example path expression which traverses to our friends in a hypothetical application named *social*.

```
{ } app { } social { } friend {f}
```

In this syntax nodes are represented by curly brace pairs, and the labels between them represent edges between the nodes. A branch in a path expression can be

represented by putting a colon following the node where the split along multiple paths occurs:

```
{ } app { } social { } friend {f}:
  name {n},
  hobby {h}
```

Plasma will utilize these path expressions in a number of ways. First, it is typical to use a path expression when the desired result is the set of endpoint nodes reached at the final node in the expression. Second, in distributed graphs it is useful to use a path expression to define an entire sub-graph, where all nodes and edges crossed while traversing the path expression should be returned, and finally it is possible to bind variables in a query statement so that new graph may be constructed from query results. In the previous examples the nodes enclosing the `f`, `n`, and `h` show how variables are bound to nodes while traversing a path. Typically these bound variables can be used in boolean predicate statements which allow for filtering of the resultant graph.

Following the standard query style used by SQL and OQL style languages, Plasma uses `SELECT` and `UPDATE` statements with `FROM` and `WHERE` clauses for basic querying. Additionally, we use the `CONSTRUCT` statement used in `SeRQL` to allow for the construction of result graphs rather than only result sequences. Typical functions for operating over query result sequences are offered: `ORDER BY`, `LIMIT`, `OFFSET`, and `DISTINCT`. The following query returns the set of my peers who also have the social application.

```
SELECT p
FROM { } net { } peer {p} app { } social
```

If this were a first query when joining a distributed system and the peer nodes in our graph were remote nodes, then the sub-query issued to each remote graph would be:

```
SELECT p
FROM {p} app { } social
```

Using a `CONSTRUCT` statement that returns graphs rather than sequences, we can build a result graph. Here we put all of our peers document files into our own doc application's sub-graph so they can be more easily accessed for later querying. This example also shows branching in the path expression using the colon at the branch point, in this case the root.

```
CONSTRUCT {doc_app} file {f}
FROM { }:
  app { } doc {doc_app},
  net { } peer {p} app { } doc { } file {f}
```

Two features that are of specific importance for the Plasma query language are that of supporting query composition and partial matches. In order to support

composition the primary property of the query language is that queries can return graphs, so that their results can be further queried. The semi-structured database community has highlighted the importance of supporting partial matches, where a query can return a set of sub-graphs where some records contain partial information.

We adopt the “?” symbol used in most regular expression languages to mean the presence of zero or one of the preceding elements. A query for our friends who optionally have a hobby would be expressed as following.

```
SELECT f, h
FROM { } app { } social { } \
      friend {f} hobby? {h}
```

The primary query feature that is unique to Plasma is the concept of recursive and iterable queries, which can be used, for example, to implement multi-hop routing protocols. These `ICONSTRUCT` and `RCONSTRUCT` queries look the same as a typical `CONSTRUCT` query, except they also allow for a stopping condition which is a boolean predicate using the `UNTIL` clause. Without an `UNTIL` clause, these queries will continue until either an empty result graph is returned or the system parameter `MAX-ITERATIONS` or `MAX-RECURSIONS` is reached. In an `ICONSTRUCT` the querying node will issue each successive query, while in the `RCONSTRUCT` successive queries will be performed on whichever node receives

A greedy network join algorithm that sought to run 3 iterations selecting the 5 peers of our peers with the closest network ID to our own would be expressed as:

```
ICONSTRUCT {p}
FROM { } net { } peer {p}
ORDER BY p.id DESC
LIMIT 5
UNTIL query.iter == 3
```

This example shows the use of a built-in *query* object, which represents a query as it moves around a Plasma network. Giving access to this object makes it simple to do iterative queries for a set number of iterations, or in this case network hops.

Beyond performing queries to retrieve result sequences or graphs, Plasma needs to provide a suite of set- and graph-oriented operators that work on graphs themselves. However, these are functional operators that are implemented as library functions, rather than being integrated into the query language. The set operators, which accept graph objects as their arguments, perform the union, intersection, complement and concatenation of two graphs. The graph operators are iteration and analysis functions which compute statistics for properties of nodes and neighborhoods of nodes. These functions can be implemented with extra logic to be efficient

over distributed graph structures, and would be used by higher level algorithms to compute properties such as the cluster coefficient of a node or neighborhood, connectivity statistics, shortest path etc.

Using a declarative query language means that the query expresses the desired information rather than the algorithm for attaining it. This allows for optimization within the query processor, but the features of the query language will not be sufficient for some algorithms and applications. In a distributed system where all peers are trusted then Plasma could serve more as a graph based distributed processing layer providing program distribution services similar to the MapReduce [5] framework. Another option is to use mobile agent techniques to safely run mobile code on remote peers. These security and functional distribution decisions are left to the user though, rather than being dictated by the Plasma framework.

4.1 Examples

In this section we present a set of examples showing the use of Plasma in distributed systems. All of the examples will be issuing queries over a set of peers that are each maintaining a graph such as the one presented in figure 1.

4.1.1 Chord

In Chord P2P networks the lookup algorithm to find an object uses a finger table which holds logarithmically spaced peers around a circular address space. Objects are hashed into this same address space so they can be stored in the peer which manages its region of ID space. The lookup here is performed recursively by doing a search using the finger table.

```

RCONSTRUCT {f}
FROM {n} chord {} finger {f} net_id {fid}
WHERE CHORD_BETWEEN(fid, n.id, my_object.id)
LIMIT 1
ORDER BY DESC(fid)

```

Note, this example makes use of a hypothetical boolean construction, CHORD_BETWEEN, which takes into account the circular address space by doing modular arithmetic.

4.1.2 Social network

In this example a user is querying for the names of the friends of their friends who enjoy skiing.

```

SELECT {n}
FROM {} friend {} friend {f} profile {}:
                                hobby{h},
                                name{n}
WHERE h == 'skiing'

```

5. ARCHITECTURE

The Plasma platform provides a graph abstraction for creating distributed programs. Developers implement

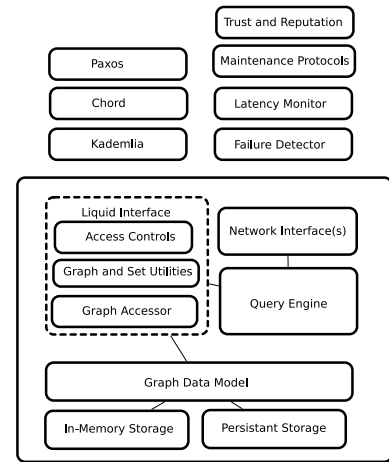


Figure 2: The architecture of the Plasma framework. Shown above are a representative set of modules which could be used to compose distributed applications.

algorithms and protocols by expressing their program's data as graph structures, and then operations are performed by using the supplied graph operator functions and by issuing queries on top of these structures. Initially the Plasma system consists of a single library that is embedded in each peer process in a distributed system, but a separate daemon process could be used to manage the graph structure and networking for multiple programs at the same time. Here we outline the architecture of the Plasma platform which is currently under development.

The design of Plasma is similar to that of an embedded database such as SQLite [6], except with additional features to support network operations. The graph layer provides a basic accessor API which allows for direct access to the graphs, nodes, and edges that make up the basic data structures. Above this layer is a collection of graph and set operators that make up the functional core of the platform.

Sitting above the basic accessor API and the graph operators is the Plasma query engine, which resembles typical database design. The query engine parses a query into a query graph, which can be translated into a query plan for further optimization at both the network and local graph algorithm stages. For example, these optimizations can seek to minimize network traffic by bundling multiple sub-queries for the same host in a single message. Moreover, the graph layer can use special data structures and indexes to minimize query processing time. The query planner provides lots of space for future work by allowing optimization to occur at many points in the processing of a query.

The networking layer is agnostic to the rest of the system, although it could interact with the query plan-

ner to offer support for prioritization of messages and QOS.

6. EXPERIENCE WITH PLASMA

Our current experience is limited to implementing example algorithms over Plasma graphs running in a library. We find that the graph layer does provide for a powerful abstraction for implementing many typical network algorithms; however, there have been a few practical issues raised when working with our current model. For example, we find that the event mechanisms, although quite powerful, can become cumbersome when a subgraph is to be monitored for events, which requires an edge from each node to the handler node. In a reasonably dynamic graph this requires constant maintenance to add these edges. One mechanism that could help alleviate this developer overhead would be some kind of view mechanism defined by a graph query, but this has not yet been explored. Additionally we find that ordering is not an easily supported concept in our current model, and the large amount of duplicate meta-data is not efficient in terms of storage or network messages. Overall we are pleased with the primary abstractions, and as of yet we have not encountered any major obstacles to further development of the Plasma system.

7. FUTURE WORK AND CONCLUSION

The Plasma model is currently implemented as a centralized simulation and a discrete event simulation, which allows us to experiment with modeling applications and algorithms over graphs. Work is underway to implement the full system in the form of an in-memory graph database library that supports the networking features described above.

The current Plasma data model is maintained in an in-memory graph that is changing often, but it is clear that integrating a graph database would be beneficial to support persistent storage. In a widely distributed system where peers are not expected to be present most of the time versioning becomes an important data storage feature, and in the future we would like to add versioning features to the graph data model as well as the persistence layer.

An open area of inquiry is to develop a security model for this type of graph based networking system. Additionally, we believe that there is a lot of interesting work to be done in developing intelligent query planners that have knowledge of graph operators, P2P and distributed systems so that they can tune performance for different operational characteristics.

Finally, we have started work on a number of software abstractions for programming with Plasma graphs. Analogous to object-relational mapping which is used to integrate relational databases with object-oriented

languages, we are working on a concept called *Perspective programming* which is uniquely graph based. The Lore [7] project developed the concept of data-guides to aid exploration of semi-structured databases, but we believe there is much more interesting work to be done in this area of program design.

We have presented a new model for distributed and peer-to-peer computing based on distributed graph structures that coordinate through the use of remote queries. In this model remote data is accessed in the same way as if it were local using graph queries that transparently cross network boundaries. Using a number of typical examples from peer-to-peer networking, distributed computing and social networking applications, we have shown that the Plasma graph data model has good expressive power while simplifying many common network tasks.

8. REFERENCES

- [1] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1–39, 2008.
- [2] C. W. Bachman. The programmer as navigator. page 1973, 2007.
- [3] J. Broeskstra and A. Kampman. Serql: A second generation rdf query language. In *SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, Amsterdam, Netherlands, Nov 2004.
- [4] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–383, 1970.
- [5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI '04*, pages 137–150, December 2004.
- [6] D. R. Hipp. Sqlite database library, apr 2004.
- [7] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.
- [8] S. B. Navathe. Evolution of data modeling for databases. *Communications of the ACM*, 35(9):112–123, 1992.
- [9] G. Noordende, B. Overeinder, R. Timmer, Brazier, F.M.T., and A. Tanenbaum. A common base for building secure mobile agent middleware systems. In *Proc. Int'l Multiconf. on Computer Science and Information Tech.*, 2007.
- [10] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, 1995.
- [11] R. W. Taylor and R. L. Frank. Codasyl data-base management systems. *ACM Computing Surveys*, 8(1):67–103, 1976.