Peer-to-Peer Programming with Teaq

Huw Evans* and Peter Dickman

Department of Computing Science
The University of Glasgow
Glasgow, G12 8RZ, UK
{huw, pd}@dcs.gla.ac.uk

Abstract. This paper introduces Teaq, a new peer-to-peer programming model and implementation that places processes into a self-healing, ordered spanning tree, across which distributed object queries are routed. The programmer has control over where in the tree their process resides, how their queries are routed through the tree, and how result objects are generated and passed back to the query initiator. Default implementations are provided that the programmer may specialise. This paper introduces the two main algorithms for maintaining the tree and routing queries.

1 Introduction

The Teaq[1] system addresses two shortcomings in the peer-to-peer literature [3]. Firstly, processes in the network are organised according to the processing capabilities of the machine on which they are running. Secondly, programming-level objects in the network can be easily found by propagating a dynamically updateable query through the network.

Peer-to-peer systems are still in their infancy. It is currently not clear what kinds of topologies the machines and processes of a peer-to-peer network should be arranged into and when one topology should be favoured over another one. Object discovery in peer-to-peer networks is also not well understood. Objects must be found in a dynamically changing, error prone, decentralized system that has a high degree of machine and network heterogeneity, as shown by [5].

The motivation for Teaq is to provide a run-time system that allows machines and processes to be arranged in such a way that the system can make profitable decisions about the shape of the underlying peer-to-peer tree topology. In addition, the system should be able to easily find programming-level objects in a network where the frequency of object creation and destruction is high. Flood routing (e.g., as used by early Gnutella systems) is wasteful of network bandwidth. Other systems such as Pastry [4] and FreeNet [2, 3] that use hash-based routing rely on the fact that the object being searched for is long-lived. Neither of these kinds of solution are appropriate in the kinds of context that Teaq is aimed at.

The rest of this paper is organised as follows: section 2 describes how processes are placed into a spanning-tree; section 3 discusses the dynamically updatable, distributed object query mechanism; section 4 concludes the paper.

* This work is funded by the UK EPSRC under grant (GR/N38114).

[1] Teaq stands for Trees, Evolution and Queries and the word is pronounced the same as teak.

2 Constructing a Teaq Tree

Teaq processes communicate events to other Teaq processes and events carry code with them. When an event is received by an object, a reference to the receiving object is passed to the event. This means that the destination object can respond to an open-ended, evolvable set of events which is not hard-wired into the receiving object. With this mechanism it is possible to dynamically update the run-time system (code and state), by replacing one object with another.
A Teaq process is an instance of a Java virtual machine and each process has within it three main objects: the first is the `Capacity` object that is used to decide where the process should be placed in the tree; secondly is the `ProcessDescriptor` object that provides a unique identifier for the process and which also abstracts over the IP address of the machine and the port number on which the `ST` object is listening; and lastly is the Spanning Tree object (`ST`) that provides management for the process once in a tree, e.g., this object receives requests from other processes that may wish to become children of this process. The `ST` object has references to the `Capacity` and `ProcessDescriptor` objects.
The `Capacity` object tells Teaq how much processing capability that process has. A process running on a machine that has a large amount of physical memory, a fast CPU, and which is attached to a fast network with a lot of bandwidth, is going to have a `Capacity` object which reflects the capabilities of this machine. The capacity object is also used to control the number of children any one process is willing to support. The `Capacity` object currently contains a single integer value which is defined on a per machine basis. The more powerful the machine, the larger the integer value will be[2].

Spanning Trees Processes form a spanning tree [1] and so have at most one parent and up to M children, where M is bounded by the capacity object. In such a tree, all processes except one (the root process) have a parent. All processes are members of the one tree and it is possible to route a message from one node to any other, assuming a route between the two is available in the face of failure (see **Failure** below). There are therefore N-1 connections between processes in the tree, where N is the number of currently involved processes. Queries are routed across this tree at run-time and the initiator of the query is seen as the root of the tree.

Connecting to a Tree In the current implementation of Teaq, the tree attachment algorithm consists of two parts: a check is first made to see if there are any local processes that may be attached to; if not, a remote process is searched for to act as the parent. In this paper, only the protocol for remote attachment is discussed in detail. Briefly, local attachment consists of one process acting as the parent for all other local processes and that parent connects to the remote tree. If the local parent fails, a leader election protocol is executed. If, when a process is started it can find no local processes, it must find its rightful place in the remote

[2] Capturing the different aspects of a machine's capabilities (e.g., the amount of memory available, the available bandwidth and even aspects such as the observed error rate) in a multi-dimensional value space, together with a more powerful comparison mechanism (that does not rely on the current total ordering of capabilities) would give Teaq more power in deciding how to adapt the tree at run-time to make best use of the available resources. This is an area for future work.

tree. To connect to the remote tree, the initiating process joins a multicast[3] group that can be seen by other processes on other machines. A datagram packet is sent to it, announcing the presence of the new process. Processes that respond send back their `ProcessDescriptor` objects. From this list, the initiating process selects a process with a capacity higher than its own. This ensures that the processes with a higher capacity, i.e., those that are capable of supporting more children, are placed closer to the root. The initiating process then sends this selected process a `ConnectionRequestEvent` that will be processed in the target process. If there is no such process, e.g., there are only two processes and the one with the lower capacity started first, the process with the highest capacity in the list is returned.

Attachment Algorithm The initiator's `ConnectionRequestEvent` is received by the target process and the attachment algorithm in listing 1.1 is then executed.

```
connecting = true

lock( st ) {
    if ( target process has available capacity )
      attach new process
    else {
      if ( target process is full and new process has less capacity ) {
            move some children of target process to new process
            attach new process to target process
      }
      else {
        if ( capacity of new process >= capacity of target process ) {
          if ( target process has no parent )
            make new process the parent of the target process
          else
            forward request to the parent of the target process
        }
      }
    }
}

connecting = false
```

Listing 1.1. Pseudo Code for the Process Attachment Algorithm

The first two if-statements of this algorithm add new processes to the tree at the first process that asserts can handle it. This may involve some manipulation of a process' children. The second two if-statements cause the tree to be extended, so that one process does not become the parent of too many children.

Maintaining the tree property is an important aspect of this algorithm and two important parts are the prevention of cycles and dealing with capacity ties. To prevent a cycle between two nodes that attempt to connect to each other at the same time, the attachment algorithm sets a boolean variable `connecting` to true before attempting to connect to the tree. When the two nodes try to connect to each other, they notice their `connecting` is true and so back off for an amount of time that is drawn randomly from an exponentially increasing value. `connecting` is also used to prevent cycles between more than two nodes. If one node attempts to connect to another node when both nodes have the same capacity (i.e., there is a capacity tie), the process that initiated the connection becomes the parent. In

[3] In the current prototype implementation of Teaq the availability of multicast is assumed. This has implications for scale as IP-multicast is not widely supported on the Internet backbone. In future versions it will be possible for the programmer to specify a set of well-known processes to connect to.

future it may be necessary to temporarily tolerate the tree property being false, e.g., in the face of multiple process failure cycles may develop. Further analysis of the tree property is an area for future work.

Failure Processes can fail as they may crash or the machine that is hosting them may be shutdown. If a parent process fails, all its children are temporarily disconnected from the tree. Should this happen, when a process next needs to communicate with its parent, an error will be received and the tree attachment algorithm, given above, is re-run.

3 Teaq Queries

Objects are found in Teaq by running queries across the tree. A query is an OQL-like statement of this form: `select t from T where t.m() == true;`. All object of type `T` whose method `m` returns true become part of the query result. A programmer makes an object available for local or remote query by registering it (and implicitly, all objects reachable from the registered object) with `ST`. The registration code builds a data structure for fast lookups based on an instance's class. When running a local query, the array of objects is returned immediately. However, in the remote case, it may not be feasible to send back to the query initiator a serialized copy of the matching instance. Instead we may want to send back an object that refers to the remote matched object. Teaq supports this by allowing a programmer to pass back a proxy to the matched object as part of the query result.

Sending a Query into the Tree To initiate a remote query, a programmer first of all creates a listener object that will be called-back whenever a query result has been found. This object is passed to the Teaq run-time system, together with a class object that indicates the type of the instances the query is interested in. The run-time system then sends out a default query object into the tree. The programmer receives an object of type `QueryToken` that represents the remote query and which defines a method called `close`. `close` is typically called by the programmer when they have received the result they require and they, therefore, want the propagation of the query to be stopped in the system (see Query Termination below).

Routing a Query Through the Tree Query routing in Teaq is flexible. A programmer is able to define how this routing is performed through a tree and what actions are performed at each process. A default query routing object is provided which, currently, visits every process, thus for a system consisting of N machines, there will be at least N messages sent out per query.
The role of the `QueryToken` object is to manage the initiating end of the query, such as receiving results and passing them to the listener. The query event that is sent into the system has three main objects that control the actions of the query as it is replicated throughout the tree: `Distance`, `Reply` and `QueryId`.
`Distance` controls how far the query will travel in the system. Currently, it ensures that the query will visit every process in the system (visiting children nodes in parallel). The `Reply` object passes results back to the initiating process.

`QueryId` is a probabilistically[4] unique query identifier. It is retained on the initiating side and a copy is kept in the query event that sent into the system.
When query results are passed back from the remote query, a copy of the `QueryId` is sent back as well. This is compared with the local copy to check the identity of the remote query. This comparison is necessary as the query could be locally shutdown and another started that just happened to be listening on the same local server-socket. In this way, the initiating side can be probabilistically sure that query results are from the query that it sent into the system.

Query Termination When the programmer has the result they desire, they call the `close` method. This shuts down the initiating side of the query. When the remote query event arrives at its next process, it will execute the query and attempt to pass the results back. However, this will not work as the initiating side has closed its side down. The remote query will receive an error message. In the current implementation, it is assumed that seeing such a message means that the initiating side has been closed down. Therefore, the event does not send itself to any parent or children processes. In this way, we prevent the remote query event from propagating itself. This removes the need to run a costly distributed query termination algorithm. However, this assumes that errors are not transient. If the query was propagated, by the time a new query result was ready to be passed back, the error situation may have passed. This approach also reduces the parallelism of remote queries but in the current prototype implementation this is felt to be preferable to having to run a distributed query-termination algorithm. The programmer can choose how to program this aspect of the system; an alternative mechanism would be to allow the query to propagate to children without having to send the result back first and to check back with the initiating site every `n` hops where `n` was configurable.

4 Conclusions

This paper has introduced the Teaq system for programming peer-to-peer systems via a self-healing, ordered spanning tree, that supports flexible routing of object queries across it. The contribution of this paper is the use of the self-healing, machine-capability-aware spanning tree to dynamically organise the underlying peer-to-peer topology and the promotion of the query-based model of programming. This paper has shown that short-lived objects and peer-to-peer programming can mix.

References

1. Mikhail J. Atallah, editor. *Algorithms and theory of computation handbook*. CRC Press, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA, 1999.

[4] It is currently a Java integer assigned at random using `java.util.Random`. This shall be changed in a future version to more accurately reflect the process that initiated the query.

2. Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hongang. Freenet: A distributed anonymous information storage and retrieval system in designing privacy enhancing technologie. In Hannes Federrath, editor, *Designing Privacy Enhancing Technologies*, volume 2009 of *Lecture Notes in Computer Science*, Berkeley, CA, USA, July 2000. Springer-Verlag, Berlin Germany.
3. Andy Oram (ed). *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, 2001.
4. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, November 2001.
5. Stefan Saroiu, P. Krishna Gummadi, and Steven D Gribble. A measurement study of peer-to-peer file sharing systems. Technical report, Department of Computer Science and Engineering, University of Washington, 2002.