Notes on efficiency of

Operational Transform in ceda

David Barrett-Lennard
Nov 2009

Confidential

Introduction

This white paper presents a brief analysis of the performance of the ceda implementation of Operational Transform (OT), and makes some comparisons based on the descriptions presented in white paper(s) provided by Google on the *Google Wave Protocol* (GWP) which also uses OT. In particular see

http://www.waveprotocol.org/whitepapers/operational-transform

Data schema

Ceda supports nested values types that resemble C/C++ structs for defining a database schema. This must be kept in mind in a comparison to the GWP which concerns operations on XML documents.

In ceda, the data is typically recorded in fields within many objects. Objects are uniquely identified using *Object Identifiers* (OIDs). Fields within a given object instance are identified by *paths*. Therefore fields of a particular object instance can be identified with both an OID and a path. This pair is called a *Field Identifier* (FID). FIDs are used in the representation of operations. Note that FIDs never need to undergo transformation under OT.

The design of a ceda database schema should normally consider the trade-off that exists between using lots of small objects compared to a smaller number of large objects. More accurately, there is a trade-off between many small fields compared to few large fields.

For example, the paragraphs of a very large text document can be stored in separate objects (or perhaps separate fields of the same object). This gives each paragraph unique identity across all sites and this can yield substantial advantages in the performance of OT. Operations on a given paragraph of text always apply to that and only that paragraph. Ceda never transforms operations against operations with different FIDs.

Ceda uses B+Trees indexed by FIDs to record information about changes to fields. Therefore logarithmic complexity is involved when searching for information about a field for a given FID.

A single XML document used in the GWP will typically map to multiple objects in an equivalent ceda database. The GWP records insertions and deletions on a given XML document using zero based index positions. Very large XML documents are more likely to have large numbers of insertions and deletions generated by many users. This increases the work required for OT under GWP.

Quadratic complexity of operations on strings

Consider the following code written in C++

```
std::vector<char> str;
const int N = 1000000;
for (int i=0 ; i < N ; ++i)
{
    str.insert(str.end(),'x');
}
```

This builds a string with a million characters. On a modern machine with gigabytes of memory and memory bandwidth measured in gigabytes per second, such a string wouldn't normally be regarded as having the potential to bring the system to its knees. In fact the above code executes in only about 40 msec on a 3GHz E8400 dual core machine.

Now consider that we make a seemingly minor change:

```
std::vector<char> str;
const int N = 1000000;
for (int i=0 ; i < N ; ++i)
{
    str.insert(str.begin(),'x');
}
```

This takes about 5 minutes to execute on the same machine! The problem is that it has quadratic performance, and a million squared is a big number. For every character to be inserted into the string, all the existing characters must be moved to make room.

It is worth keeping this in mind when discussing the complexity of algorithms on XML documents (for GWP) or for text fields (for ceda). As a rough rule of thumb, it is a bad idea to allow contiguous strings in memory to exceed about 10 kbytes if characters can be inserted at any position, In ceda the appropriate action is to break the data into small pieces by virtue of an appropriate schema design. In GWP specialised techniques may be warranted, if XML documents can become large.

OT on a single text field

Ceda records operations on a text field using an ordered linked list of intervals. Ceda doesn't (directly) implement the inclusion transform. Instead it *merges* two operations using an algorithm that is very analogous to the merge of two sorted lists as used in the mergesort algorithm. The merge of two lists of sizes n and m involves linear scans through both lists and therefore has complexity O(n+m).

Google claim their algorithm is O(n log n + m log m).

OT on assignable fields

Ceda provides a very simple and efficient method for OT on assignable fields. It is essentially O(1) because for each assignable field (i.e. FID) a site only records the current value of the field plus a vector time. Irrespective of all prior assignments to the field, the sender only sends the current value and the vector time. OT only

involves lexicographical comparison of vector times to determine the winner. If the remote assignment dominates the local assignment then the local field value and vector time are assigned with the remote values.

It is not actually clear how the GWP will support assignments. Presumably it must represent assignments as delete + insert of a textual representation of a value recorded in an XML document. However concurrent assignments would be expected to result in multiple values in the resultant XML document (since deletes can never erase concurrent inserts). Possibly the dominating value will be the first in a list of values.

Ignoring this difficulty, concurrent assignments on unrelated fields are transformed against each other whenever the fields are recorded in the same XML document. In fact these operations contribute to the n,m in the O(n log n + m log m) complexity of the GWP.

It is assumed that the GWP approach manages to compress causally dependent sequences of assignments to the same field. If not, the effect would be that many assignments to the same field increase the values of n.m. If compression is not performed then there also will be substantially increased storage space and bandwidth consumption compared to ceda.

OT on sets and bags

Ceda treats these fields specially and therefore would be expected to offer substantial benefits over stringwise operations on XML which imposes an ordering where no such ordering is warranted.

This is particularly relevant to the relational model (noting that a relation is a *set* of tuples).

GWP and waiting for ACKs

The GWP has the following requirement: After a client has sent a 'chunk' of operations to the server it must wait for an ACK from the server before it is allowed to send the next chunk of operations.

Google state:

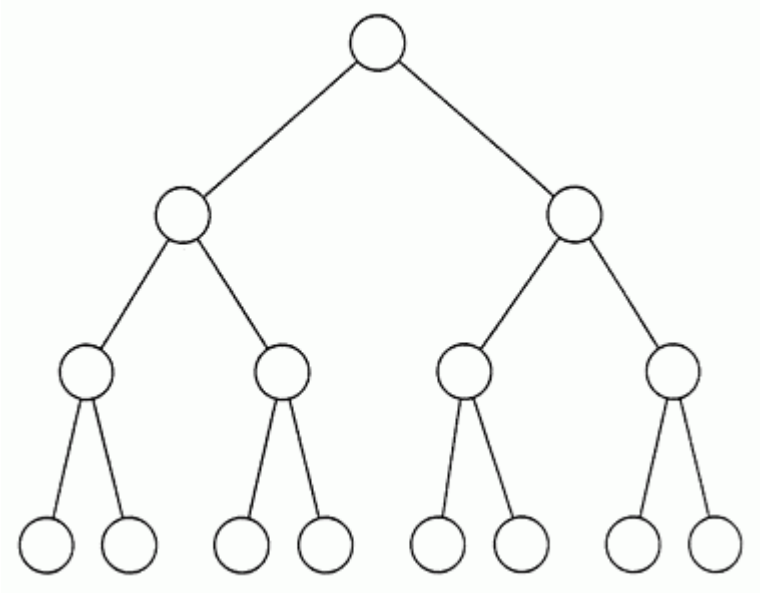
> *One trade off of this change is that a client will see chunks of operations from another client in intervals of approximately one round trip time to the other client. However, we believe the benefits obtained from the change make this a worthwhile trade off.*

This approach prohibits the streaming of fine grained operations from a client. E.g. if a client is moving an object with the mouse and generating operations at the rate of 50Hz, and if the *Round Trip Time* (RTT) to the server is 200 msec then the maximum rate at which operations can be streamed to the server (and hence to other clients) is actually only 5Hz, irrespective of the bandwidth available. The result is that the objects will not appear to be smoothly manipulated on other machines.

In some applications there may be users that are interested in watching work being performed like a movie. In such cases a consistent delay is tolerable, but a poor update rate is not. For example seeing the work performed by a talented graphic artist could be upset if operations are sent in coarse chunks. Note as well that ping times as high as 400msec are not all that uncommon on the Internet, so the update rate could fall below 3 Hz with the GWP.

Network topology

Ceda supports very large peer-peer networks. There can be many machines that contribute to the effort of merging operations. In a sense it is like a distributed merge computation. Consider a network topology based on a binary tree:



While off-line, let the leaf nodes all together generate a grand total of n insertion operations on the same text field. In the above diagram there are 8 leaf nodes so we assume each generates n/8 operations.

Note that while a leaf node is off-line, the locally generated insertion intervals are ordered as they are generated. A simple approach is for each new interval to be inserted into the existing list in sorted order based on a linear scan to find the insertion position. Such an approach is like insertion sort and has quadratic complexity. That is not necessarily something to be concerned about - after all, the execution of those same insertion operations on the contiguous text field has quadratic complexity anyway. In any case one could avoid the quadratic complexity by recording the intervals in a red-black tree (for example).

We will analyse the subsequent merging effort by the nodes assuming all leaf nodes go on-line at the same time. For simplicity we will assume that operations are always passed up towards the root node in single bulk transfers, advancing one level at a time, and then the operations propagate back down again, again using bulk transfers, advancing one level at a time, until reaching the leaf nodes. At that point all the leaf nodes should have converged to the same final state.

Each leaf machine transmits sorted insertion intervals up to its parent node when it goes on-line. In a binary tree a given parent will receive two such sorted lists from its two children and merge these into a single larger sorted list using simple linear scans in the manner of the merge step in a merge sort algorithm. The merged list of intervals is passed up to the next level in the binary tree.

At each level in the tree the same grand total number of operations (n) need to be processed. The machine at the root is most overloaded for operations that are merging going up the tree, because it must single-handedly process all n operations coming from below.

Going back down again, each machine will need to merge all operations received from above that are not already present on that machine. For example each of the two nodes immediately below the root will need to merge n/2 local operations with n/2 operations received from above. i.e. a total of n operations must be processed. At the next level down there are n/4 local operations and 3n/4 operations received from above. Again we see that each machine must process all n operations. Evidently even each leaf node will need to process all n operations in order to merge the operations that are received from above.

We have the curious result that the most heavily loaded machines are the two that sit just below the root, because these machines must process n/2 operations upwards, and n operations downwards. This is a total of 3n/2.

The conclusion is that the work load is remarkably even across all nodes - including leaf nodes. Furthermore, the work load is linear in the total number of operations that have been generated across the system, which is the best that one could hope for.

The merging process going up towards the root node is reminiscent of disk based sort algorithms that sort very large lists (that cannot fit into memory). This is performed in multiple passes where each pass corresponds to recursive calls to merge sort considered in bottom up order (as the stack unwinds). In ceda a given machine can be regarded as running a “merge pass” on all the operations it receives “below it” and forwards a single merged stream as an output to some machine "above it". Remarkably this merging effort can be performed in different directions at the same time (a machine can act like a ‘hub’ – forward on operations in different directions). It is readily achievable because the sending of an operation involves a linear scan for the intervals that need to be forwarded on in a particular direction.

It is unknown to this author whether there is any counterpart in the GWP. Presumably it is possible to use a hierarchy of servers instead of a single server. To what extent is it expected that end user machines will be able to take on some responsibility for the merging work required by a hierarchical configuration of servers? It seems unlikely given that the GWP cannot support arbitrary network topology changes. In practise end user machines are comparatively unreliable (at least in terms of connectivity on the network) so it could be dangerous to use any of them as servers (i.e. so they help define the “linear history path” that must be defined collectively by a hierarchy of server machines).

Ceda Repository

A *ceda repository* is used for managing complex data, and supports branching, checkin, checkout, tagging and merging. Ideally check-ins are associated with well defined high level tasks. Since only a few check-ins per day are expected from a given user, it is clear that a repository can easily cope with the workload for thousands of users working on massive projects. Note furthermore that ceda repositories can be distributed (another benefit of an approach that satisfies TP2).

Live collaboration is always *relative* to some state vector (or "branch") defined in a repository. This helps enormously with performance, because:

1. Latecomers are brought up to date quickly by retrieving the initial state of a text field from a repository (and without the need to generate the state from executing many tiny operations)
2. There are no users that perform operations in a context prior to a relatively mature state recorded in the repository. Therefore during the live collaboration there is no need to transform operations in a mature state of the text field against concurrent operations performed in a much earlier state.
3. During the live interaction, insertion intervals on a text field are only recorded relative to the state of the text recorded in the repository, therefore there are far fewer intervals.

The effect therefore is that the n,m tend to be small in an O(n+m) merge. It is expected that eventually a live interactive mode will be closed, and a single user will commit all the changes as a single, optimally compressed delta to a repository.