Robust network topologies

for replication in CEDA

David Barrett-Lennard

19 Aug 2013

## Introduction

This paper investigates approaches for having a set of computers (called *nodes*) connect in an efficient topology that is resilient to network and machine failure.

It is assumed each node listens on a single port to allow for stream oriented connections initiated by certain peer nodes (typically using TCP). There is a *Topology Working Set (TWS)* which is replicated over all nodes and contains information about the topology and health of the nodes and the network. This is used to both control and monitor the network.

The CEDA implementation of Operational Transformation (OT) allows for this working set to be replicated and synchronised very efficiently. It is helpful that CEDA's implementation of OT satisfies TP2, so any two nodes can connect, exchange deltas and synchronise at any time, whether or not they have connected before in the past. Therefore there are no constraints whatsoever on changes to the topology that may be utilised as a result of network or machine failures, the introduction of new nodes into the system etc.

## Desirable characteristics of the topology

The *connection graph* means the set of nodes plus set of edges between pairs of nodes, where an edge means the two nodes establish a TCP socket connection between them in order to support bidirectional exchange of data.

We assume we are talking about a single connected network - meaning that it is desirable for the graph to have at least one path between every pair of nodes. In graph theory one would say the graph is *connected*.

Note that the connection graph is defined on top of the underlying IP routing. For efficiency it is desirable for the connection graph to mirror the physical topology in some sense, but how that is achieved is outside the scope of this paper.

It is assumed that ideally a connection graph doesn't contain any cycles, because a cycle means there is a redundant path for the transmission of synchronisation deltas and this represents suboptimal use of the available network and processing bandwidth. In CEDA this leads to redundancy ("overlap") in the deltas that may be received by a node. It doesn't cause synchronisation to fail - it's just wasteful so we avoid it. It follows that the connection graph is ideally a *tree* (in graph theory, a *tree* is an undirected graph in which any two vertices are connected by exactly one simple path).

In CEDA the protocol for sending and receiving deltas between a pair of connected nodes is perfectly symmetrical, which would suggest one can think of the connection graph as an undirected graph. However, in reality we need to make a distinction just for the purpose of establishing the connection in the first place. We expect one node listens on a port, and another node initiates the connection using the known hostname and port of its peer. Because of this lack of symmetry it is useful to consider the connection graph to be a directed graph. An easy way to specify a direction on all the edges of a tree is to nominate a root and to consider all the edges to be directed towards or away from the root.

In graph theory the diameter of a graph is the maximum distance between two nodes. We define the diameter of the connection graph as the maximum number of hops over all pairs of nodes. For obvious reasons it is desirable to minimise the latency in the system and therefore minimise the diameter of the graph. The diameter of a full n-ary tree is twice the height of the tree.

Let the *fan-out* of a node be the number of child nodes. The *maximum fan-out* which is appropriate to a node depends on the network and computational bandwidth available to the node. The number of nodes grows exponentially with the height of the tree. If the maximum fan-out is F then the number of nodes N for a tree of height h is given by

 N = $\sum\_{i=1}^{h}F^{i}$

So for example, for F=10 a tree with at most 12 hops allows for 1+10+100+1000+10000+100000+1000000 which is over a million nodes.

The CEDA implementation of OT is linear in the total number of operations generated across all nodes. As a rule of thumb a modern machine can process roughly a million simple operations per second generated by peers. The aim is to support networks with up to about 105 nodes, so we want to be careful about when and why operations are applied to the TWS because it should only represent a tiny proportion of the processing capabilities of a node. It is expected that normally most of the nodes in the network are connected and working properly and not generating operations on the TWS.

## Network partitions

Usually connectivity between hosts is reflexive, symmetric and transitive and therefore can be modelled as a partition of the network into mutually exclusive groups. IP routing succeeds between any two nodes in the same partition and fails between any two nodes in different partitions.

We expect the system to form an appropriate tree of connections within each network partition. Once network connectivity is re-established we expect the network topology to change accordingly.

The system needs to be resilient to any and all failures with nodes and the network. For example, there is no special dependency on the root. If it fails then another node automatically takes on that role as required.

## Nominal versus actual topology

The *nominal topology* is defined with respect to the current physical hardware in the system on the assumption that all the hardware elements are functioning correctly. The nominal topology is recorded in the TWS. This paper is not concerned with how to define an appropriate nominal topology for given physical hardware, and whether the nominal topology is derived and updated automatically or else by human input. In practise it is expected that the nominal topology is fairly static. For some systems, particularly where the nodes are servers it will be defined by system administrators, perhaps through a GUI which allows for viewing and editing the data in the TWS. The replication of the TWS means that updates can be applied to any node in the system and the changes automatically propagate to the other nodes.

The *actual topology* refers to the topology that the system is using given various fault conditions with nodes or the network. It is expected that this is more dynamic in nature, although hopefully faults with the system are quite rare in practise, so the actual topology coincides with the nominal topology. This paper is concerned with how the system achieves an appropriate topology for arbitrary faults.

## NodeIds

The nodes in the system are allocated *node identifiers (NodeIds)*. We expect that the system will tend to grow over time by adding nodes to an existing topology. It is assumed there is a total order defined on NodeIds. This allows an order to be defined on the child nodes of every node. In graph theory this is referred to as an *ordered tree*.

There are two options we will consider:

1. Assume NodeId allocation is centralised, giving out identifiers using a sequence number starting at 0. It follows that arrays can be used to record information about the nodes using a node identifier as an array index.
2. Use 128 bit GUIDs for the node identifiers, allocated in a decentralised manner.

## The Topology Working Set (TWS)

The TWS records information about all the nodes in a map keyed by NodeId. This for example records the hostname and port for each node. It is assumed these hostnames have been recorded correctly and each node will listen as a server on the port number so recorded. Any node can obtain the hostname and port to initiate a connection to a peer identified by its NodeId.

Information about the topology of the nodes is recorded by specifying the parent node of every node. This will be NULL\_NODE\_ID for the root node. It is easy to calculate the children of each node using a single linear scan over the array of nodes.

$typedef+ Guid TNodeId;

const TNodeId NULL\_NODE\_ID = Guid();

// A time interval expressed in integer number of nanoseconds.

// Range is roughly +/- 292 years

$typedef+ int64 TNanoSec;

$model+ TNodeInfo

{

 TNanoSec PingFromParent; // Time for a ping initiated by the ActualParent

 TNodeId NominalParent; // The peer to which this node is nominally supposed

 // to connect or NULL\_NODE\_ID for the root node of the network

 TNodeId ActualParent; // The peer to which this node has actually connected

 int32 Cost; // The cost of the link to the nominal parent

 xstring Hostname;

 uint16 Port;

};

$model+ TNetwork

{

 // Time between each ping initiated by a parent node

 TNanoSec PingInterval;

 // If PingFromParent is within this factor of being correct then avoid

 // performing the assignment operation to update its value.

 float32 PingUpdateErrorFactor;

 // The root node of the network

 TNodeId Root;

 xmap<TNodeId,TNodeInfo> Nodes;

 int32 NumNodes;

};

## Cost factors on edges of the nominal topology

The nominal topology records a *cost factor* on each parent-child edge. This provides the basis for nested groupings of the tree structure. As an example consider three Local Area Networks shown in purple, green and orange:

1

0

0

0

0

0

0

0

0

0

0

0

1

Level 0 tree

Level 1 tree

1

1

On the left the *level 0 tree* is depicted and the nodes are actual machines. The links within a LAN are shown in black and are allocated cost 0, while the two red links which are between LANs are allocated cost 1. The effect is to partition the underlying tree into a coarser granularity *level 1 tree* (depicted above on the right). The level 1 nodes represent a partition of the level 0 nodes into mutually exclusive groups.

We can in turn have links with cost 2 which allows grouping of the level 1 nodes into level 2 nodes. This pattern can be repeated for ever coarser groupings of the nodes. All the nested groupings are at once recorded using an integer cost on each edge in the level 0 tree.

## Ordering of the children of a node

Only the parent of each node is recorded in the TWS. That suggests that no ordering of the children of a given node is defined. However an order is imposed according to the ordering of the node identifiers.

This allows for defining *depth first traversal order* of the tree of nodes under a given node:

1

2

3

7

6

5

4

10

9

11

8

Labelling of the nodes according to depth first traversal

Within a level 0 tree corresponding to a single group at level 1, depth first traversal represents the priority given for the nodes to take on the role of the root node of the group. So for example if nodes 1,2,3 all go down then we expect the remaining nodes to form a tree rooted in node 4, and node 4 will behave like node 1 for the purposes of initiating a connection to the parent group at the granularity of level 1.

## Selection of node to connect to

Robustness arises from an algorithm used by all the nodes to allow efficient trees to be formed in each network partition despite temporary or prolonged network or node failures.

Normally each node (except the root of the network) will initiate a connection to its *nominal parent*. If that can't be achieved then an algorithm defines the priority order for initiating a connection to an alternative node which takes on the role of its *actual parent*.

We will be referring to the order in which a node will attempt connections to its peers, as though it actually tries them in this order. For performance an implementation may instead attempt connections in parallel and give priority according to the order, or have more efficient ways of determining the potential for connectivity to hosts in the presence of network partitions. In this section we are only concerned with defining the priority order.

Consider the following picture of a level 0 tree that forms a single group with respect to level 1 (so all the links in this picture have cost 0):

0

1

2

3

7

6

5

4

8

10

9

12

11

14

13

We have arbitrarily picked one of the nodes (the purple node labelled 0) in order to discuss the connection attempts it will initiate on its peers. Firstly it will try to connect to its parent. Therefore this has been labelled 1 because it is the first node it will try. If that connection attempt fails (perhaps after a timeout) then node 0 will attempt a connection with node 2 and so on.

The priority order of connection attempts is determined by a traversal involving depth first traversals performed consecutively up through the linear list of ancestors. In this example it begins with a depth first tree traversal beginning at the parent of node 0 (i.e. node 1), but avoiding traversal back into node 0 or into any of its siblings to the right of node 0. This gives nodes 1,2,3 shown in red. Then we move into the grandparent of node 0 (i.e. node 4) and perform another depth first traversal, this time avoiding traversal into the parent of node 0 or into any of its siblings to its right. This gives nodes 4,5,6,7,8 coloured green. This pattern is repeated moving into the great grandparent of node 0 to give the nodes 9,10,11,12,13,14 coloured orange.

Note that nodes 1 to 14 are the nodes that are traversed before node 0 in depth first traversal downwards from node 9 (but in a different order) which is the root of this level 1 group. This is consistent with the claim that depth first traversal represents the priority given for the nodes to take on the role of the root node of the group.

Now consider that an attempted connection to all of these 14 nodes has failed. Then node 0 will assume it must take on responsibility of initiating a level 1 connection on behalf of node 9. Therefore it will attempt a connection into the parent group at the granularity of level 1. To achieve this it will look up the parent of node 9 at level 0, and use the same algorithm of depth first traversals up through the ancestors of the parent of node 9 for prioritising connection attempts to nodes within the parent group. If that fails then this traversal algorithm is applied at the granularity of level 1 (and so on for higher levels). i.e. we can speak of connection attempts at the granularity of level 1, using this same idea of a sequence of depth first traversals up through the lineage of ancestors.

## Introducing new nodes into the system

It is proposed that a new node introduced into the system is configured with the nominal parent node it should connect to. It connects to that machine and gets a copy of the TWS. It generates an entry for itself in the map and this is propagated to the other nodes.

## Responding to changes in the topology data

A node should monitor the field 'NominalParent' and if it changes then it should disconnect from the existing parent if any and reconnect to the new nominal parent.

It seems appropriate to create a dependent node of the Dependency Graph System which is associated with the calculation of the set of nodes which have equal or higher priority that the node it happens to have established a connection to. If this set changes it can indicate a need to break the existing connection and form a new connection with some other node.

## Pinging

Each node is responsible for sending a ping message to each of its children at the rate determined by the PingInterval. Each child node is responsible for sending back a ping response message as soon as possible. A parent node should update the PingFromParent if it is different from the current value by more than an error factor (which is a setting recorded in the working set). It should set -1 for children that aren't connected or that don't respond in an appropriate timeout.

## Recording the actual parents

Information about connectivity is recorded using assignments to the ActualParent field. This is assigned as follows:

* If after a time out has elapsed and no connection to a parent has been successful then the ActualParent of itself is assigned NULL\_NODE\_ID.
* If after a time out has elapsed and no ping message has been received from a child which has ActualParent set to this node, then the ActualParent of the child is assigned NULL\_NODE\_ID
* If and when a node establishes a connection to a parent it assigns the ActualParent of itself

A node can traverse the ActualParent links upwards and downwards from itself to determine the partition of nodes it exists in. This can be useful information when a node is a server that is responding to a request from a client with a forwarding address to an appropriate server to handle the request. It is appropriate to favour servers which are known to be online according to the connectivity information.