First publ. in: IP.com Technical Disclosure IPCOM000018579D, July 2003

Synchronizing Entities in Replicated Log-Structured Databases with Disconnected Operation

Marcel Waldvogel

July 2003

Abstract

In distributed and/or replicated systems, there is often the need for one system to get updated information from others. For this reason, there is a node notation introduced, in which each node has a unique ID of an ID tree. The root node (the first node in the system) has a zero-length name. Whenever an additional node joins, it gets the name of the introducing node as a prefix, followed by a unique symbol which the parent has not yet appended.

In distributed and/or replicated systems, there is often the need for one system to get updated information from others. This is complicated when, e.g.,

- number of nodes is very large,
- not every node reliably knows all others (e.g., during join/leave situations),
- some nodes are not always reachable (disconnected operation or network problem),
- the nodes are connected through an overlay network which may change its topology or contain unreliable nodes.

There are well-known optimal solutions for some of the trivial cases, such when the number of peers is limited (e.g., maintaining a backlog for the unreachable peers), there is only a single node (or small number of nodes) which create(s) new information (e.g., keeping a table with the highest sequence number of a change for each node), or when the number of documents kept at each node is small (e.g., hashes or hash trees).

The present idea provides a new means for distributing and synchronizing information efficiently when the above adverse issues are present. It additionally requires some transport protocol as a means for distribution; practical mechanisms might include epidemic-based rumor spreading or distribution through an overlay network.

The solution is based on the following considerations:

- Every piece of information (including every change to this information) is associated with a unique "localized sequence number", consisting of a tuple (*nodeID*, *sequenceNumber*). *nodeID* is the globally unique identifier of the node that created this information. *sequenceNumber* is a monotonically increasing sequence number in the domain of the *nodeID*, i.e., each update performed by our node gets assigned the next higher sequence number. Sequence numbers of different nodes have no direct relationship, each node's sequence number indicates the number of updates this node has injected into the database/network.

- Each node will try to spread any new information (whether locally generated or remotely received) to some other nodes according to the transport mechanism.

- When two nodes enter into a "relationship" (aka "synchronization connection"), they compare their set of (*nodeID*, *sequenceNo*) tuples and request/transfer the information that is missing on either side.

This mechanism does not work well when the set of (*nodeID*, *sequenceNo*) tuples is large, as transmitting the information to be exchanged when entering into a relationship is potentially huge.

The key to our solution is to have the node IDs to follow a specific pattern, as follows:

Each node has a unique ID of an ID tree. The root node (the first node in the system) has a zero-length name. Whenever an additional node joins, it gets the name of the introducing node as a prefix, followed by a unique symbol which the parent has not yet appended.

In the simplest case, these symbols are of fixed length, say eight bits, each represented here by a character, for readability. The root node has name " ". The nodes that ask it for a name get names "a", "b", "c", etc. When "b" is contacted by a new node, it starts handing out names "ba", "bb", "bc", etc. These names will never conflict with names given out by other nodes, and each of the nodes can add members without ever contacting any other member.

If symbols are of limited size, as in the example given above, we run into a problem when a single node is asked to hand out more names than it has distinct symbols. There are several ways around this problem:

Tell the requesting node to contact another node for the new name (disadvantage: requires communication and at least one member with available IDs to be online; by definition, there will always exist at least one member with available IDs, but it may not currently be online).

Use variable-length symbols, where the symbol space may be extended by prefixing with a special marker, or where an implicit or explicit symbol length is given (examples: (1) when "bz" would need to be assigned by "b", it instead starts assigning "bza" or "bzaa", where the "a" or "aa" part again can be incremented as needed; (2) use the most significant bit of each byte to indicate whether this is the last byte in the symbol or not; (3) have a symbol-terminating

marker, e.g., a NULL byte, (4) have each symbol start with its length (e.g. "1b2af1d3kfm")).

When replicating/synchronizing with a neighbor, the tree of nodes may be more efficiently encoded, similar to the DNS structure, where pointers to the parent may be more explicitely given. A joining node should chose among the parents with shorter IDs to keep the tree somewhat balanced (the performance of the relationship protocol depends on the maximum depth of the tree).

When two nodes enter a relationship, instead of exchanging the entire table, they only exchange the hash tree, and only those parts that are different. (A tree-node in the hash tree that is identical among both partners indicates that all the children are identical.) The hash tree is built as follows, according to the *nodeID* tree (in other words, it is a tree of the *nodeID*s, where the branching factor is k, which corresponds to the number of symbols given out by the node controlling that trie-node). There are two variants:

Variant 1

- Each leaf tree-node's value corresponds to its *sequenceNo*.
- Each non-leaf tree-node's value is a hash function of its child(ren) tree-node(s) (sub-)name/*sequenceNo* pairs, ordered by name (subnames are the part of the name that is needed to distinguish this node's children).

Variant 2

- Each leaf tree-node's value is a function of its name and the *sequenceNo*.
- Each non-leaf tree-node's value is the hash function of its children, ordered by name.

After this exchange phase, it becomes apparent which informations are missing on which side. The (expected) performance of this algorithm is $O(d \cdot b \cdot \log N)$, where b is the average branching factor and d is the number of *nodeID*s that have gained information on either side, assuming the tree to be somewhat balanced. When almost all nodes have changes to report, it degrades to the basic list mechanism.