The Mercurial wire protocol

Mercurial performs all of its network transactions over HTTP or SSH.

See also the HttpCommandProtocol and the SshCommandProtocol.

Overview

Discovery of Common Changesets

A couple of functions are used to efficiently discover the boundaries of set of nodes common to both the client and server, without a massive number of round trips or needing to send the entire changelog list.

There are two versions of this today. Set-based discovery is used against newer servers and generally uses fewer roundtrips. It discovers the heads of the common subset. Tree-based discovery is used against older servers and discovers the bases of the incoming set, which is less precise.

Transmitting Changesets

Then there are functions to apply or obtain bundles of changesets. The latter are again present in older versions (changegroup*) that use bases, and a newer version that uses heads of common (getbundle).

Newer Set-based Protocol

Wire functions used:

- `heads()` - Return a list of heads (everything new must be an ancestor of one of these heads, so start here).
- `known(list)` - Return a string of 0/1 indicating whether or not each of the node ids in the list is known to the server.
- `getbundle(heads, common)` - Return all changesets reachable through nodes listed in heads (tracing ancestors) minus those reachable through nodes listed in common.
- `unbundle(bundle)` - Apply the changesets in the bundle to the server's repository.

Gist of how discovery of the common heads works:

- Get server's heads. If all are known locally, we're done.

- Define three sets of nodes:
  - common: local nodes known to be common; initially empty
  - missing: local nodes known to be missing on the server; initially empty
  - undecided: local nodes where we don't know which other set to put them into yet; initially the entire set of local nodes
- Update common and undecided by moving everything reachable via ancestors through one of the server's heads that we know locally to common.
- Take an initial sample of the remaining undecided set.
- Call known(sample).
- Update common and undecided by moving everything reachable via ancestors through one of nodes in the sample known by the server to common.
- Update missing and undecided by moving everything reachable via descendants through one of nodes in the sample unknown by the server to missing.
- While undecided is not empty, take a full sample and continue above where we call known(sample).

For the initial sample, we first add the heads and then walk the undecided set from its heads towards its roots, taking samples at exponentially increasing distances. We stop once the sample is full. If it is not full at the end, we fill it with a random sample of the unsampled nodes. The idea is that this is quick (only ancestor information needed, does not necessarily walk the entire graph).

The full samples walk from both the heads to the roots and vice versa. And they always walk the full graph. If the final sample ends up too big, we take a random sample of it. Since we only do this after the first quick sample has (hopefully) already reduced the undecided set considerably, we ususally don't have to invert a lot of the graph for descendant information (which is costly). We sample from the roots too for cases where both the client and server have a lot of mutually unknown nodes.

Older Tree-based Protocol

The network protocol looks like this:

`heads()`

```
 return a list of heads
 (everything new must be an ancestor of one of these heads, so start
here)
```

`branches(list)`

- list = a list of tips
- tip = the last revision in a branch that we're interested in (often a head).
- base = the first revision going backwards from the tip that has two parents (the product of a merge).
- p1 = the first parent of the base.

- p2 = the second parent of the base.

```
 for node in list:
   follow the linear section of a branch back to its branchpoint
   return (tip, base, p1, p2)
   (this reduces round trips for long linear branches)
```

`between(list)`

- branch = Basically a linear set of changes. More formally, a list of changes such that no change in the set (except possibly the last) has two parents, and all changes in the list are related to eachother by parent->child relationships.
- list = a list of (tip, base) tuples for branches we'd like to know the history of. Presumably the client knows about the base of the branch, but not the tip, and wants to find out where in the branch its knowledge ends.
- tip = the latest revision in a branch the client is interested in (may not be the actual tip of the branch on the server side)
- base = the earliest revision in a branch the client is interested in. Always a revision the client knows about.

```
 for tip, base in list:
   walk back a linear branch, return elements 1, 2, 4, 8..
   (and this lets us do bisection search if we diverge in the middle of
one of these long branches)
```

`changegroup(roots)`

- roots = a list of the latest nodes on every service side changeset branch that both the client and server know about.

```
 find all changesets descended from roots and return them as a single
changegroup
```

Changegroup Format

A changegroup is a single stream containing:

- a changelog group
- a manifest group
- a list of
  - filename length
  - filename
  - file group (terminated by a zero length filename)

A group is a list of chunks:

- chunk length

- self hash, p1 hash, p2 hash, link hash
- uncompressed delta to p1
- (terminated by a zero length chunk)

Changegroups are encoded in BundleFormat.

Hgweb/remoterepository currently runs this all through zlib which makes sense on WANs, but less sense on LANs.

CategoryInternals

WireProtocol (last edited 2011-05-01 12:19:06 by PeterArrenbrecht)