USING REPLICA SERVERS TO OFFLOAD QUERIES

How Polyhedra's read-only replicas can be used to reduce the workload on the main database servers, and achieve better overall performance and resilience.

Abstract

The Polyhedra in-memory DBMS comes with a read-only replica mechanism that can be used to offload some of the work of query handling from the main database server. This paper discusses some of the configurations that are possible, and the benefits to be gained by the use of the replica mechanism.

Contents

ENEA

Replica configurations

Simple server (no replicas)

For completeness, we shall show the configuration file needed to set up a simple Polyhedra server with no replicas. This will act as a starting point for the more-complex configurations in later sections. Basically, the database server (the rtrdb executable) needs to be told what port (or ports) client applications can use to connect to it, and the name of the file that holds a copy of the database: this file will be read on start-up, and over-written when the server is told to save the database or shut-down the service. (For safety, the old version of the file will be renamed fore a new snapshot is written.)

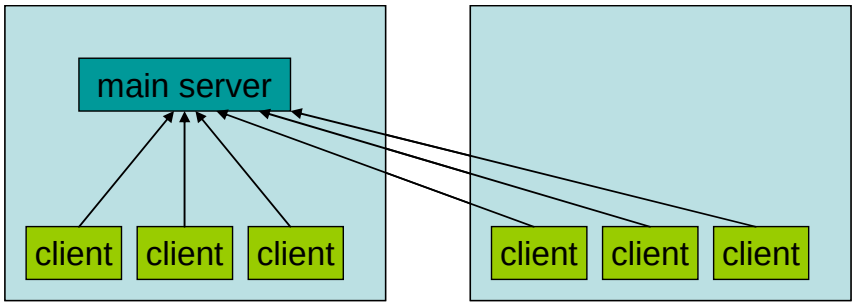
These configuration parameters can be supplied via command line parameters when starting the server – but in practice it is better to store them in a configuration file (call, by default, poly.cfg). If this file contains the following lines:

```
common:
data_service = 8001

empty:common
type         = rtrdb

db:empty
load_file    = test.dat
```

… then the entry point called ‘empty’ inherits the resource setting ‘`data_service=8001`’, and defines the type resource to have the value rtrdb; the ‘db’ entry point extends this further by adding in a definition of the `load_file` resource. If the server is set of by means of the command ‘rtrdb empty’, it would look up ‘empty’ in poly.cfg; the type resource acts as a safety check (the ‘empty’ entry point is suitable for use with the rtrdb executable), and the data_service resource tells it the TCP/IP port to claim for incoming client connections. As the load_file resource is not defined for this entry point, the database starts up empty; if, instead the server had been started up using the command ‘rtrdb db’, the server would have known where to look for a database file.



Polyhedra allows client applications to be local to the database server to which they are connecting, or on a different machine. There is no need for the separate machines to be running the same operating system – or even for them to have the same underlying hardware architecture, as the Polyhedra client-server connections automatically handle issues such as differences in endianism. It is also possible for a client to have multiple connections, to different database servers.

Simple replica

If you want to be able to set up a replica of a database server, the first thing you need to do is to tell the main database to allow replica connections. This is done by setting up a resource that specifies the port to which replicas are to connect. For example, if we add the following lines to the configuration file described above…

```
dbj:db
journal_service = 8051
```

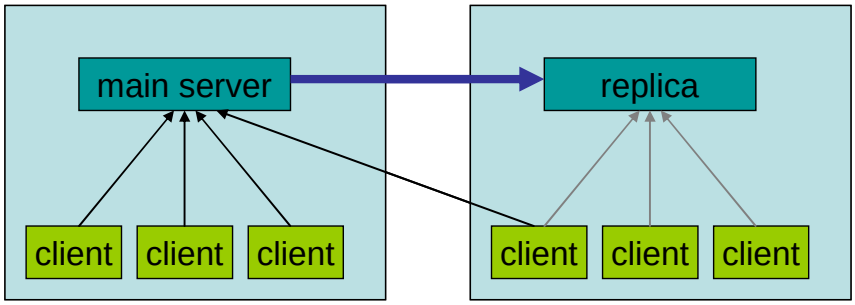
… then starting the Polyhedra database server with the command 'rtrdb dbj' instead of 'rtrdb db' means that the server will allow a replica to connect to port 9001. To start the replica, you simply tell the server what port it should serve for client-server connections, and the journal service port of the database it is to replicate:

```
common:
data_service = 8001

dbr:common
type                = rtrdb
replication_service = 192.168.121.101:8051
```

Assuming the database to be replicated is on the machine with IP address 192.168.121.101, then on the machine that is to run the replica you can just go to the directory containing a poly.cfg file with the above contents, and issue the command 'rtrdb dbr'.

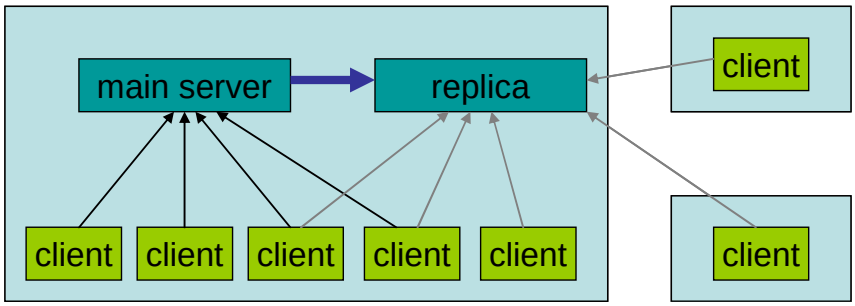
When the replica starts up, it connects to the journal service port of the main server, and is given a complete copy of the database. It will then be sent a stream of journal records, which it can apply to its copy of the database. A heartbeat mechanism lets the main server know that the replica is still alive, to avoid a backlog of journal records building up if the network or replica has fallen over.



By the way, the Polyhedra client-server libraries will not allow client application to connect to the replica simply by quoting the port address 8001 (prefixed by the machine address and a colon if running remotely), as a a replica is essentially read-only and by default the client libraries will only connect to a read-write server. To get round this, you can either append the <ro> option to the service name when opening the connection ("8001<ro>"), or the <replica> option, or – for complete generality, the <any> option, viz "8001<any>". (The list of options and their meanings is described in section 4.1 of the Polyhedra user guide, as part of the description of the data_service resource.) As a Polyhedra client application is allowed to have multiple connections, it can have a read-only connection to the replica that it uses for queries, and a separate connection to the main server that it uses for updating the database.

The main advantage of running a simple replica is that of query offload. The replica can be running on a different machine to the server holding the database to be replicated, and thus processing queries does not steal CPU cycles from the server holding the main copy of the database. This can make the overall system more responsive, particularly when a client has launched a 'killer query' that might take seconds to process. Replicas also allow 'query localisation': the server being queried is closer to the client application, which reduces

latency. Depending on the balance between queries (from the clients on the remote machines) and database updates, it can also reduce the network traffic.



It is of course possible to install the replica server on the same machine as the main server.[1] Unless they machine has lots of cores, this can mean there is some contention for CPU cycles – and also for memory cycles, depending on the size of the database, the computer architecture and the size of the cache(s). It does, however, still have the advantage that the main database server can be processing updates even while the replica is processing a killer query involving the tables being updated.

If there are lots of clients wanting to launch queries, you can have a number of replicas that will share the load between them.

Replica of a master-standby pair

Polyhedra allows a pair of servers to be set up in a master-standby configuration, to give resilience. An arbitration mechanism is provided so that (customer-supplied) software can instruct a server which mode it is to start in, and later instruct the standby to go to master status. (At this point, the old master – if it is still running – will automatically stop so that it can be restarted in standby mode.) To start a server in fault-tolerant mode, one simply has to tell it how to connect to the local arbitrator component, the port it is to use for the other server to connect to it, and the journal port of the other database server. Thus, if one of the machines has an IP address of 192.168.1.151, you could start the server on the other machine with a command such as 'rtrdb dbft' if you had added the following lines to the configuration file:

```
dbft:db
arbitrator_service    = 7200
arbitrator_protocol   = TCP
journal_service       = 8051
other_journal_service = 192.168.121.101:8051
```

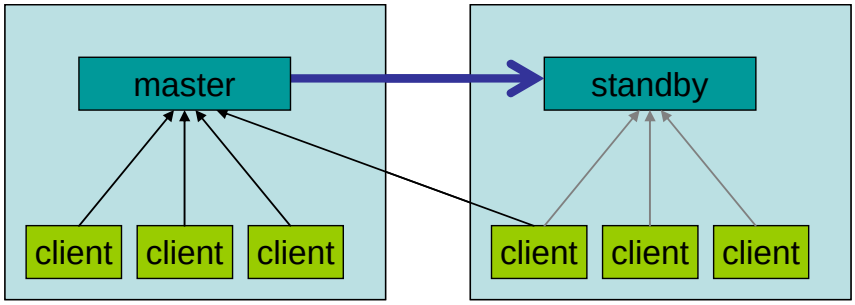
(Another way of achieving the same effect would be to omit the other_journal_service line from the configuration file, and instead pass it across via the command line:

```
rtrdb -r other_journal_service=192.168.121.101:8051 dbft
```

This has the advantage that the poly.cfg file can be identical on the two machines, at the cost of having to use different command scripts on the two machines for starting the servers.)

[1] Of course, it will be necessary to ensure the replica server was using a different port for client connections than that used for the main server!

From the above, you can see that the standby connects to the master using the same port as is used for replica connections, and is updated in a similar fashion: it gets a complete snapshot of the database, and then a stream of journal records to keep it up to date. The main difference is that the standby also sends back a stream of acknowledgements so that the master knows when transactions have been 'made safe'.



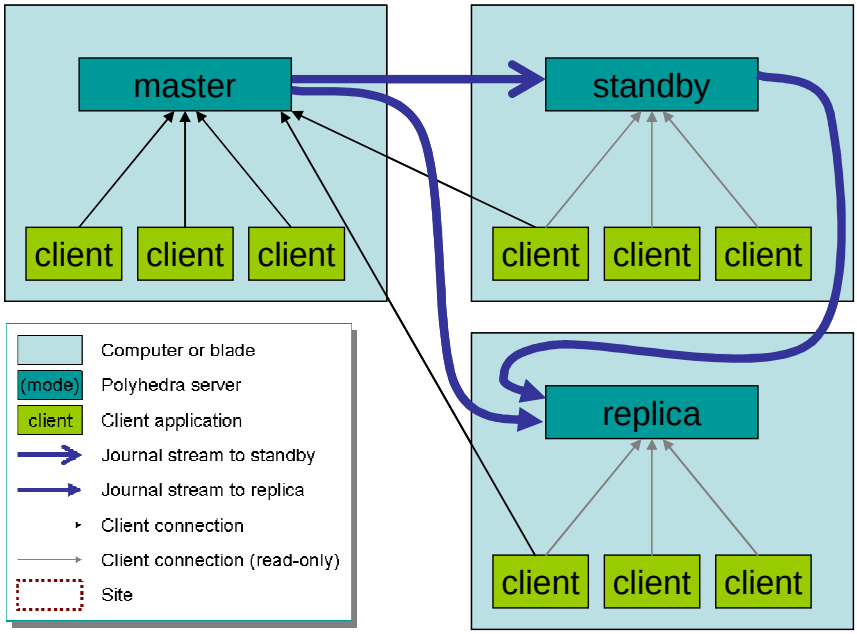
As with replica connections, the standby is read-only (until it is promoted to master), so clients choosing to connect to it have to confirm that they really want to talk to that server – typically, by using the [any] option. (Clients can also ask for a fault-tolerant connection, in which case they have to give a list of access points; the library will connect to the master, and will automatically fail over to the newly-promoted server if a failover occurs.)

If you have a fault-tolerant database service, it is natural to consider having a replica of the database that knows about both servers. This can be done very simply: instead of telling the replica about one of the servers via the replication_service resource, you give the address and port number of both journal service access points:

```
dbr:db
replication_service = 192.168.121.101:8051,192.168.121.102:8051
```

When the replica is started, it will try to connect to the servers in turn, trying to find out which one is master. It will then get a complete copy of the database from the master, and then listen to the stream of journal records.

It can now try to connect in a similar way to the standby. However, in this case, though, it does not get another copy of the snapshot, merely a stream of information about journal records that have been applied to the standby. This allows the replica to avoid 'getting ahead' of the standby: while it is connected to both the master and the standby, it delays applying journal records until it knows the transaction has been applied to both servers. This allows it to handle server failover more cleanly, as it never has to undo a transaction that the master told it about but which had not been fully applied to the standby at the time the master failed. Of course, if the standby is the server that fails, the replica merely has to close the connection, apply any pending journal records it has received from the master, and then periodically try to reconnect to the standby.



Replica of a replica

It is possible to set up multiple replica servers connecting into the same stand-alone database server, or into a fault-tolerant pair of servers. However, there is an overhead on a server for each connected replica: it uses more memory and CPU cycles, and of course the amount of I/O increases. To improve the scalability, then, Enea has added a mechanism that allows a fan-out of servers. Thus, is the primary server is running on machine 192.168.121.101, you can set up a secondary server on machine 192.168.121.202 with the command 'rtrdb dbr' if its local configuration file looks like the following:

```
common:
data_service = 8001

dbr:common
type                = rtrdb
replication_service = 192.168.121.101:8051
journal_service     = 8051
```

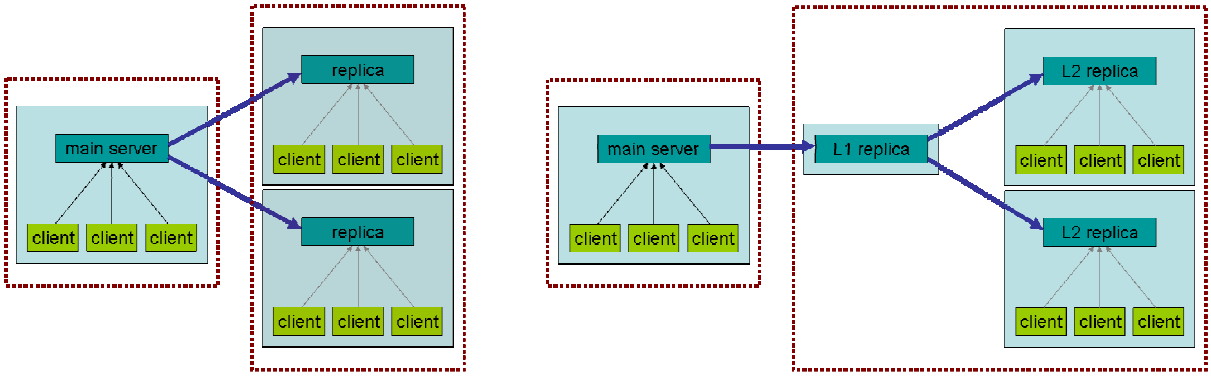
When compared to previous examples you should see that his has an extra line in the dbr section, defining the journal service access point; this instructs the replica to set up an access point that allows a 2nd-level replica to connect to it. A suitable configuration file for that one, assuming it is running on a separate machine, could be:

```
common:
data_service = 8001

dbr:common
type                = rtrdb
replication_service = 192.168.121.202:8051
journal_service     = 8051
```

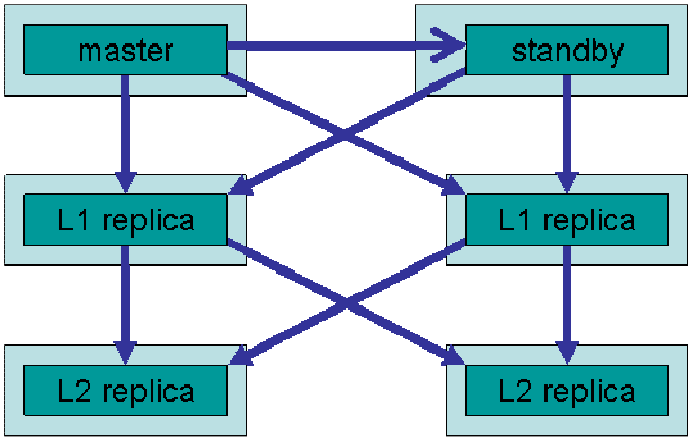
The last line can be omitted if you do not want to set up a 3rd-level replica. Polyhedra imposes no limit on the length of a replica chain, though of course the servers further down the chain may noticed increased latency!

Multi-level replication allows you to set up replica farms with minimal load on the main server, as it is only concerned about keeping the first-level replicas up to date. It is particularly useful when multiple replicas are wanted on a remote site: in the diagram below, the configuration on the left is conceptually simpler, but the traffic over the relatively-slow inter-site link will be much reduced in the configuration on the right – and the overall latency will only be marginally greater.



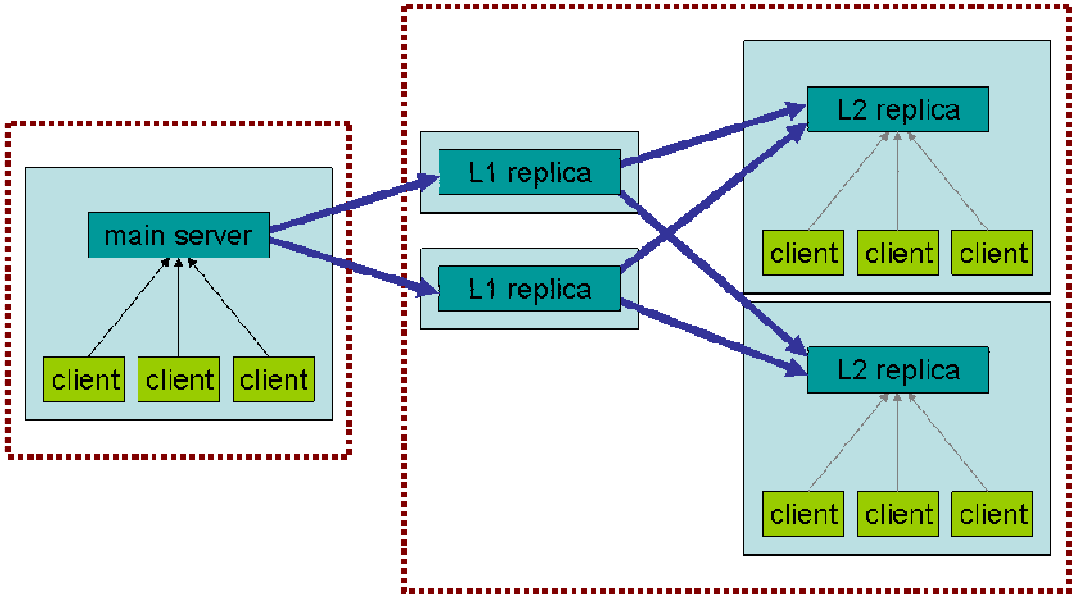
Replica of a pair of replicas

It is possible to set up a 2nd-level replica of a server that replicates a fault-tolerant pair – and as the 1st-level replica survives a failover of the database service, the 2nd-level replica will also survive. However, if the 1st-level replica fails, the 2nd-level replica will stop, so the service it offers is not fault-tolerant. The way round this is to configure the 2nd-level replica to monitor two 1st-level replica servers – so if one of those fails, the 2nd-level replica can continue based on the information coming from the other:



Provided one of the master or standby is still running, and one of the 1st-level replicas is still running, then (depending on network connectivity) each L2 server should see an unbroken service.

It is also possible to set up a fault-tolerant replica of two replicas of a server that is not itself fault-tolerant. This could be useful if you have a remote site connected via two separate network links, and you want the remote replicas to survive the failure of one of those links:



Summary

The Polyhedra IMDB servers provide a read-only replication mechanism that can be deployed in a variety of configurations, ranging from the very simple set-up to a complex fault-tolerant fan-out configuration that can survive single-point failures if used on a resilient network. This flexibility does of course mean that some thought might need to be given to find the optimal configuration, but in general where there is a heavy query load (or where some client applications will be launching particularly complex queries) it is straightforward to deploy one or more replicas to improve the overall responsiveness of a Polyhedra-based system.

For more details of the Polyhedra® product, please visit our web site, www.polyhedra.com or www.enea.com/polyhedra, or email us at info@enea.com