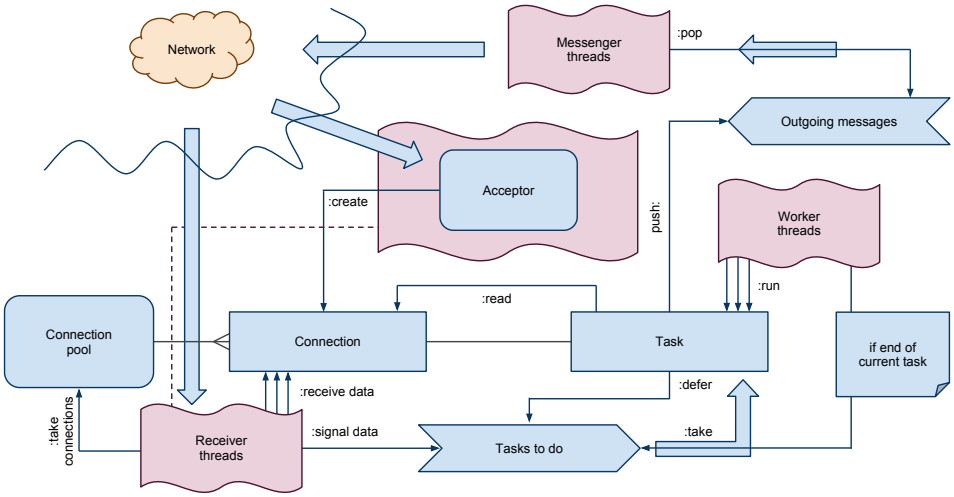
CoherentDB Net server project

Cezary Bartoszuk Rafał Rawicki Michał Stachurski

November 8, 2010



1 Threads

The basic description of threads working in the system.

1.1 Reveiver thread

```
ReceiverThread:
    loop:
        myConnections := ConnectionPool.takeMyConnections(self)
        for activeConnection in select(myConnections):
            activeConnection.pull
```

Acceptor is also run at a *receiver thread* all the magic is hidden in the `pull` method of the connection object. Described below.

1.2 Worker thread

Supposing *Tasks to do* queue is a monitor the pseudo-code looks similar to this:

```
WorkerThread:
    loop:
        task = tasksToDo.pop
        task.run
```

1.3 Messenger thread

Similar to *Worker thread*, but gets data from *messages* queue.

```
MessengerThread:
    loop:
        message := outgoingMessages.first
        message.send!
```

The data burst implementation will be probably hidden in here.

2 Queues

Both *Tasks to do* and *Messages* are monitor-synchronized queues.

2.1 Tasks to do queue

A FIFO (or maybe Priority?) queue. A monitor with standard producer-consumer idiom:

```
monitor tasksToDo:
    var buffer: Queue
    var size: Integer
    var full: Condition
    var empty: Condition
    method push(item):
        while size = MAX_SIZE:
            full.wait
        buffer.push(item)
        size := size + 1
        if size = 1:
            empty.notify
    method pop:
        while size = 0:
            empty.wait
        item := buffer.pop
        size := size - 1
        if size = MAX_SIZE - 1:
            full.notify
        return item
```

The C++-specific implementation will probably be parameterizable with a queue (so that any data structure with `pop` and `push` - or similar - operations will be apropriate).

2.2 Messages queue

Probably the same idea as *Tasks to do* queue - e.i. producer-consumer pattern with parameterizable queue-like buffer inside.

3 System-wide objects

One object, exactly...

3.1 Connection pool

Provides useful API for the *Receiver threads* to watch the connections for the incoming data.

Implements the `takeMyConnections(t: Thread): Collection[Connection]` method which ensures that at the moment only one *receiver thread* will read one connection object.

4 Per-session objects

4.1 Connection

Represents a network connection.

Should implement methods:

- `pull` for retrieving data from the connection (if it's the special connection object - the acceptor - create new connection and register it into the connection pool).
- `active` if a data waits on the connection and a task is waiting on the data.

Record items:

- `DataObserver` triggered when apropriate amount of data comes in.

Getting data is kind of *lazy*. We buffer incoming data in the application only if a task is waiting on the data.

4.2 Task

An object representing a callable.

There are two kinds of tasks:

- Read-waiting tasks
- Deferred tasks

Read-waiting task is a callable that is triggered whenever apropriate amount of data comes in. It's a two parameter function. The first parameter is the handle for the incoming data. The second one is the context in which the function will be called.

Deferred task is a callable, which running was postponed for later. A task can defer many tasks of which every has to either send a response via the messages queue or join in a Join point.

4.3 Join point

An object which triggers running a task *after* the last of specified tasks is done. The continuation task may be run in two ways:

- Eager
- Lazy

The eager manner just runs the triggered task in the same thread as the last of forked threads.

The lazy method defers triggered task to the tasks queue.