Guide to txDelta Concepts and Configuration

Prepared by: Gerhardus Muller

Date: 14 May 2009

Version: 1.3

Introduction

txDelta is the a new generation mserver – mserver v2.0. The code base is the same, several new features have been added and a number of bugs removed. The name derives from tx (transaction) and delta – the way a river parallelises and splits into multiple streams at the sea. A concise set of goals for txDelta are:

1. server infrastructure to execute tasks in a controllable manner
2. the ability to separate different execution streams and monitor / control them separately
3. logging and recovery of failed tasks. Recovery to support fine grain task selection
4. monitoring and control of system resources – either the machine it is executing on or other resources such as the database
5. support to limit the effective lifetime of a request and the maximum execution time
6. monitoring of performance including queue lengths and times requests are queued before being serviced
7. support for delayed execution of events
8. dynamic control of critical server parameters without interrupting processing
9. controlled and clean shutdown on request when required

Architecture

The txDelta master process starts up and controls its child processes. It will re-start any process that dies and for the rest owns all the IPC objects. It initially spawns the socket process and dispatcher processes. The socket process provides a Unix domain, a TCP and UDP socket interfaces to the outside world for the submission of new tasks. These are passed via an internal socket to the dispatcher process for execution unless it is a control packet in which case it could be dispatched differently.

The dispatcher can be configured to start up a number of queues. A queue is either a regular queue (FIFO based) or a delay queue (which holds events until they are ready for execution and then inserts them into the same queue as a regular queue). A queue owns a dynamically configurable number of worker processes. These are fed when available with events from the queue to execute in its own process space. Events can have an expiry time associated – this limits the usable lifetime of events and they are discarded if they expire in the queue. If due to resource limitations or unavailability events are not serviced in time they can be expired instead of consuming precious resources at a time when it most likely cannot be afforded. Expired events are not written to the recovery log. A queue also has a maximum execution time attribute and a task is killed and written to the recovery log if it exceeds this time. If a stream based connection is used (either TCP or Unix

Domain Sockets) the submitting application can keep the socket open and receive the reply back from the worker once it has executed.

Installation

Refer to the document faxFx2/serverInstallation/packages/txDelta.txt

From a pre-built binary it is deployed to the target machine with:

```
./serverInstallation/deploy/txDelta.pl -C host
```

On the host it is then deployed from the txDelta/scripts directory (as root). It will not trash an existing config or log files:

```
./install
```

A manual /etc/init.d/rcX.d link has to be made. On Debian this is:

```
cd /etc/rc2.d/
ln -s ../init.d/txDelta S90txDelta
```

Start the new copy:

```
/etc/init.d/txDelta start
```

Verify there are no errors in the logs:

```
cd ../tests
./test.pl -t4
```

Testing

A test suite is available in the *tests* directory. Consult the file *testing.txt*. It is not normally required to run these tests.

Configuration

Configuration by default lives in /usr/local/etc/txDelta.cfg

Parameters that can be configured are obtained by:

```
/usr/local/bin/txDelta -h
```

In general most of the defaults are sufficient. A typical sample configuration will be discussed in more detail below. A lot of the parameters are self explanatory.

```
[main]
runAsUser = uucp
statsUrl = http://utilities/faxfx/mserverStats.php
statsInterval = 0
```

```
#statsChildrenAddress = gertdeb,deploy
#logBaseDir = /var/log/txDelta
nologconsole = 1
```

statsInterval and *statsChildrenAddress* are used to generate synchronised stats that can can be aggregated between different copies of txDelta. For standalone use only *statsInterval* is required and typically is set to 180s. If not a single copy of txDelta is configured to be master, its *statsInterval* is setup and the children it needs to notify. The children or slaves have *statsInterval* set to 0.

nologconsole is normally 1 in a production environment where the server runs as a daemon. Same holds for *logstderr* – instead of logging to stdout it logs a copy to stderr. Again not for production environment.

Although the base log directory (/var/log/txDelta) can be changed and the pid and log file names it is not in general required as it is not expected to have more than a single copy to be running on the same machine. As such all configs and logs have been move to Linux standard locations with the logs landing in a subdirectory of /var/log, the config to /usr/local/etc and the executable to /usr/local/bin.

The more interesting config parameters all relate to the dispatcher – the process that queues and farms out the work:

```
[dispatcher]
defaultLogLevel = 5
maintInterval = 5
expiredEventInterval = 5
statsUrl = http://utilities/faxfx/mserverStats.php
queue0.name = default
queue0.numWorkers = 3
queue0.maxLength = 800
queue1.name = msnreq
queue1.numWorkers = 2
queue1.maxLength = 100
queue2.name = perl
queue2.numWorkers = 4
queue2.type = delayQueue
queue2.maxExecTime = 30
queue2.maxLength = 100
```

For logging the name of the log file can be changed – by specifying and complete path it will be created outside of *main.logBaseDir*. The *defaultLogLevel* controls the amount of logging and has a default value of 5. A smaller value logs less and a larger value more. Values of 8-10 are not recommended for production use.

The *maintInterval* value sets a timer that runs maintenance tasks on a regular basis. This includes checks for expired events (if used), tasks overrunning max execution times (if configured), queue stats (controlled by *bLogQueueStatus* – a log file entry – not the full http version) and the delay queue. *expiredEventInterval* should be a multiple of *maintInterval* and determines how often the queue is scanned for expired events. Do not set any of these values shorter than required as the overhead of continuously walking a long queue could kill the processing resources on the box.

A dispatcher process can own a number of queues. In general each queue should be selected and configured for particular tasks. This allows the queue parameters to be tailored to the task at hand

and the statistics produced are then specific to the workload. Unless you have very good reason and understand the implications always configure a standard queue with name 'default' with at least one worker. This queue is used by default for events that don't explictly specify a queue and for other system events such as http reporting.

Queues should be defined one per section named [*dispatcher.queueN*]. Required parameters are *name* and *numWorkers*; *type*('eventQueue'), *maxLength*(500000), *maxExecTime*(0), *persistentApp* are optional. *N* starts at 0 and *type* can be 'eventQueue' or 'delayQueue'. Queues cannot be created dynamically. The queue name is used when submitting events for processing. The number or workers is self explanatory and can be configured dynamically without interrupting processing (same holds for *maxLength* and *maxExecTime*). If more events are queued than specified in *maxLength* the queue performs an emergency dump of its contents into the recovery log and proceeds with normal operation. *maxExecTime* if specified limits the execution time of a worker. Initially the spawned task is sent a SIGTERM signal. If on the next maintenance interval the task has not yet exited the worker itself is sent a SIGKILL which cannot be ignored. A new worker is forked. In both cases the failed event is written to the recovery log with reason either SIGTERM or SIGKILL.

The *persistentApp* parameter specifies a persistent application that is started up per worker on the queue. Please see the discussion below.

Worker parameters mainly relate to the configuration of the Perl interpreter (which is fairly standard) and the parameters which are used to determine success or failure of its spawned tasks. These parameters are tightly integrated with the current code and should not be changed. Worker parameters are global and not per queue.

```
[socket]
defaultLogLevel = 6
```

Most socket parameters are fairly self expanatory and do not need to be adjusted. It is worthwhile noting here that the hosts file needs an entry against its non-local (non 127.0.0.1) address for its configured hostname (/etc/hostsname). This is used to determine on which IP address to listen. txDelta will not listen on 127.0.0.1.

Log Analysis

Log analysis can be done with the script txDeltaLogGrep.pl. Execute:

```
txDeltaLogGrep.pl –h
```

for options. It takes a search term as parameter for which it retrieves all the log keys (related log events). All log lines for the keys are then extracted. The -s option allows extraction only of the execution result messages for quick reference if required.

Recovery

Although old style recovery is still be supported it is not recommended. Please use the new external recovery script in the */usr/local/bin/txDeltaRecover.pl*. It is far more functional and in future may be expanded to support more complex features such as rate limited recovery.

>txDeltaRecover.pl -h

```
/usr/local/bin/txDeltaRecover.pl options
        search criteria are combined with AND logic by default unless -l is
           specified
        -f recovery log file
        -r recover the selected events
        -o object type (serialisation name) of objects to be recovered
        -g grepterm - on its own it displays records, with recovery it selects
        -c select only events of this class
        -e select only events of this error type
        -l change option combination logic to OR
        -s recovery server name if TCP is used
        -p recovery server service or port (default 'mserver')
        -u Unix socket path (default to use - defaults to
           '/var/log/txDelta/txDelta.sock')
        -S do not summarise
        -d to list event detail
        -v for verbose output and listing of selected events
        -h for this help screen
```

A couple of notable script features:

1. the script can recover the live system without needing to logrotate
2. it allows for the selection events to recover based on the AND or OR combination of the event class (*-c*), error or recovery reason (*-e*) and an arbitrary pcre search term (*-g*) or the object type (*-o* serialisation name)
3. the list of events to be recovered can be previewed by initially omitting the recovery switch (*-r*)
4. the *toString()* version of the failed event can be viewed as part of the log line by requesting detail (*-d*)
5. the events can be recovered via a different server using TCP and the *-s*, *-p* switches

Dynamic Configuration

Please refer to the cmdDispatcherConf sub in test.pl in the tests subdir for a good example. Exact definition is available in the reconfigure() function in cppLib/dispatcher/dispatcher.cpp. The following commands are supported:

updateworkers, parameters *queue* and *val*

updatemaxqueuelen, parameters *queue* and *val*

updatemaxexectime, parameters *queue* and *val*

The script *txDeltaAdmin.pl* provides these commands along with the ability to shutdown the server:

```
txDeltaAdmin.pl options command command options
        -s recovery server name if TCP is used
        -p recovery server service or port (default 'mserver')
```

```
        -u Unix socket path (default to use - defaults to
'/var/log/txDelta/txDelta.sock')
        -v for verbose output and listing of selected events
        -h for this help screen
Commands:
        shutdown
        maxqueuelen queue maxval
        maxexectime queue maxexec
        numworkers queue numworkers
```

Submission of Tasks or Events

The Perl scripts in perModules:

```
Event.pm
DispatchEvent.pm
DispatchRequestEvent.pm
DispatchNotifyEvent.pm
DispatchScriptEvent.pm
DispatchCommandEvent.pm
DispatchResultEvent.pm
EventArchive.pm
```

serve as reference in terms of generating the appropriate serialised object strings. The Perl code is preferred over the C++ as it is explicit in terms of the type of parameters archived and also does not require intricate C++ knowledge of overloaded operators.

The test programs in the test subdirectory – test.pl and genEvents.pl serve as good examples of how to generate DispatchCommandEvents (*cmdShutdown* and *cmdDispatcherConf* subs in *test.pl*) and DispatchNotifyEvent, DispatchRequestEvent and DispatchScriptEvents (*generateEvent* sub in *genEvents.pl*).

A mirrored implementation is kept maintained in PHP. Both these implementations verify the protocol version to confirm that the implemented version matches the server connecting to. The PHP implementation can be found in platformV3/php/txDelta:

```
event.inc.php
dispatchEvent.inc.php
dispatchRequestEvent.inc.php
dispatchNotifyEvent.inc.php
dispatchScriptEvent.inc.php
dispatchCommandEvent.inc.php
dispatchResultEvent.inc.php
eventArchive.inc.php
```

Persistent Applications

txDelta supports a concept of a persistent application. This is an application that is started up by the worker (one copy per worker) and remains live for the duration of the server unless it exits in which case it is auto started again. There are a number of reasons for this model. The first is the ability to create additional server applications that are fed by a dispatcher queue and not tightly integrated into *txDelta* to the point that it is linked in. Rather it is spawned as an external application. It has all of the normal benefits from being restarted if it does exit or crash to being fed by any one of the queue types in *dispatcher*. Other benefits include efficiency – it runs the whole time and is not spawned for every event. It also allows for the seamless use of languages such as Perl that is often more appropriate than C++. Finally it allows an app to retain state between servicing requests. If this is the model a single worker should be configured. If more than one worker is configured (to make use of multiple cores) the app would have to resort to standard shared memory IPC.

The persistent application is expected to receive a *dispatchEvent* derived object on its input and produce a matching *dispatchResultEvent* object for every input object. It is not a requirement for it to produce a *dispatchResultEvent* - rather at least a *dispatchEvent* derived object. If a *dispatchResultEvent* is used the *bSuccess* field will be used to determine success and should be set appropriately. In this mode the persistent script should write nothing to its *stdout* other than *dispatchEvent* derived objects. *stderr* output will land in the dispatcher log. Please use explicit executable types as the execution type is derived from the extension. *.pl* (perl) *.sh* (shell) and the rest are regarded as binaries. The configuration file command line can be followed by space separated parameters. Parameters can be quoted with `"` (double quotes). '\' is the escape character for an embedded `"`. If the application cannot produce a *dispatchEvent* derived result it should exit otherwise the feeding queue will hang around and lockup the queue. Failed events (those for which the application exits or the *bSuccess* field is false) are written to the recovery log.

The application can be coded (and most likely should be) to receive and process *dispatchCommandEvent* events of type *CMD_PERSISTENT_APP*. A convention is to add (addParam) a name/value pair with key *command* and the value representing the action. Additional name/value pairs can be defined based on the command. To upgrade code one of the commands could cause the app to exit – it will automatically be respawned. Bear in mind the application should always respond with an event – event if it discards the event.