TMF Trace Structure

Version 0.3
François Chouinard

1. Introduction

This section deals with the handling of arbitrarily large event traces and how they are merged into tracing experiments.

Most of the complexity comes from large log/trace files that can't be fully held in memory and need to be traversed and accessed efficiently. The proposed solution involves bookmarking the event stream and using an event cache.

The TMF Trace class structure is sketched in Figure 1.

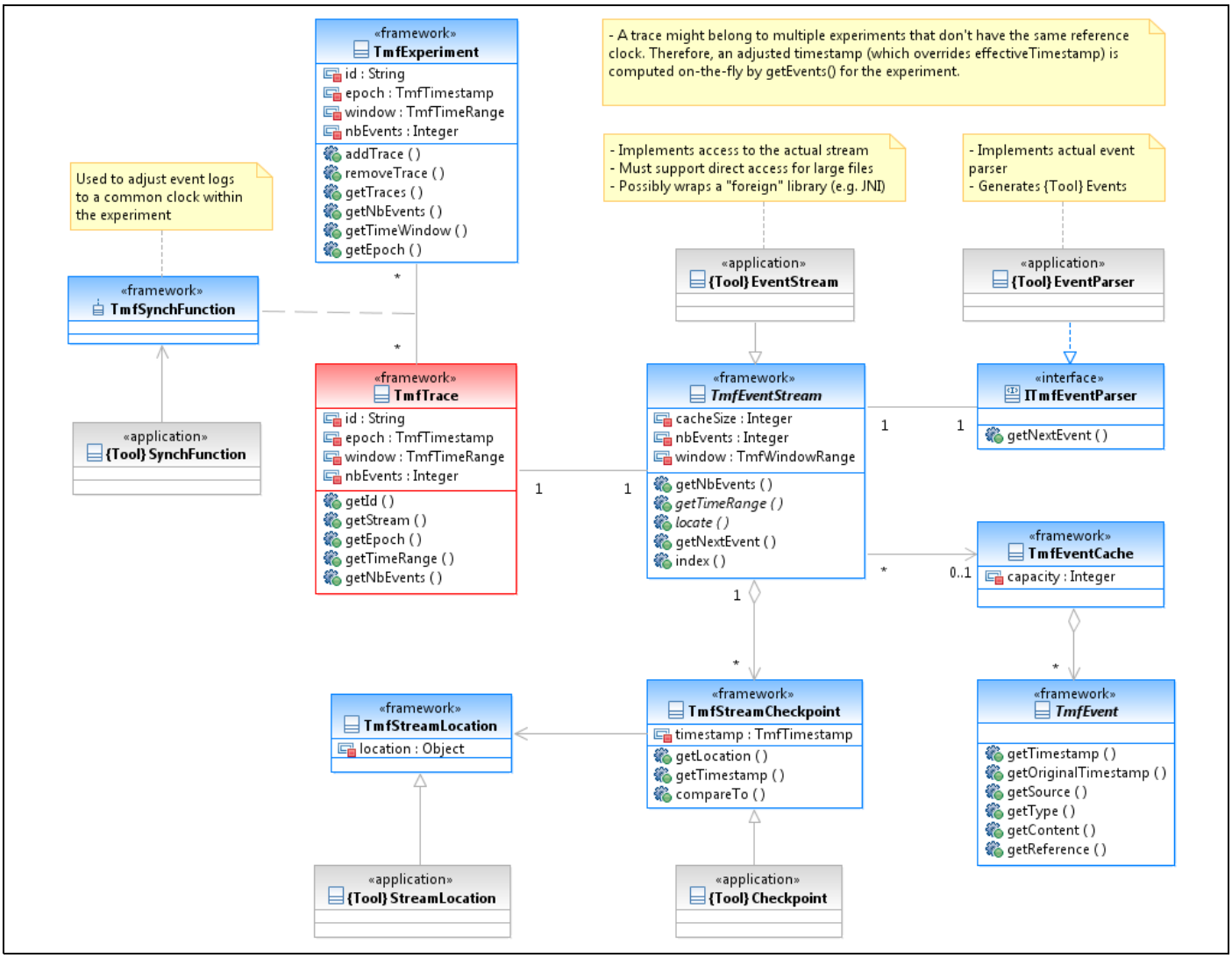

Figure 1 TMF Trace Class Diagram

2. TmfExperiment, TmfTrace and TmfSynchFunction

The *TmfTrace* represents a time-ordered (on the *effectiveTimestamp* field) set of events tied to a single event stream and keeps track of the global information about the event log:

- epoch : the reference clock for the trace events (i.e. *t0*)
- time range : the time span between the oldest and the most recent event
- nbEvents : the total number of events in the trace

Since it is assumed that the full trace can't be held in memory, it is associated to a *TmfEventStream* that provides access to the events on an effective timestamp base.

TmfExperiment provides an aggregated view of *TmfTraces* and adjusts the effective timestamps of the resulting *TmfEvent*s set to some reference clock. A *TmfExperiment* can contain multiple *TmfTraces* and a *TmfTrace* can belong to multiple *TmfExperiments*.

To realize the timestamp adjustment, a synchronization attribute, *TmfSynchFunction,* is used to qualify each Experiment-Trace association. This adjustment can be a simple value to account for clock drift or it can be a more elaborate function (this part is not yet well elaborated).

3. TmfEventStream, TmfEventParser and TmfEventCache

The *TmfEventStream* handles the actual interface with the event stream. It is an abstract class that is intended to be sub-classed, via an adapter, to handle the actual stream.

It defines methods to perform direct access to the underlying stream and provides a checkpoint scheme to quickly access any timestamp in a large stream. For efficiency purposes, checkpoints are inserted at *cacheSize* intervals.

When the stream is read (*getNextEvent()*), it returns parsed events. The application must provide its parser that generates *TmfEvents* (or extensions of *TmfEvents*) from the current location in the stream.

For efficiency purposes, the concrete stream adapter should provide a direct access mechanism on the stream. The framework will be delivered with a set a adapters for common stream types (random-access file, socket, ...)

Also for efficiency purposes, the stream maintains an event cache that acts like a window on the event stream. Note that, assuming the actual stream is not tampered with, the cache content should never become stale i.e. events are immutable and do not change in time.

4. TmfStreamCheckpoint and TmfStreamLocation

The *TmfStreamCheckpoint* is a mechanism that enables quick access to selected parts of the log file, on event boundaries.

The framework provides a very basic implementation where an event timestamp is associated to a stream location. Note that it assumes that the stream events effective timestamps are time-ordered (i.e. they are increasing monotonically).

In the basic implementation, a *TmfStreamLocation* is simply an offset into a random-access file. Considering that some streams might be rather complex (e.g. multiple files), the location structure can be extended.

TmfStreamCheckpoint can also be extended if more advanced information needs to be saved along (e.g. a global state so advanced parsing can be resumed from essentially anywhere in the stream).

5. Streaming

Although not very hard to implement, the framework does not yet provide generic support for handling streaming logs i.e. non file-based traces and logs (e.g. a monitoring application).

Therefore, it is the responsibility of the application to store streaming logs locally and implement access methods.

This feature will eventually migrate into the framework.

6. Life Cycle

Coming soon to a wiki near you.