Components

by

Table of contents

1. Scoping Appender

The heart of ProfDog is a custom log4j appender that looks for logging messages of a certain format. These messages come in pairs marking the start and end of (what we have called) scopes. These scopes have names and are nested within each other. The scoping appender takes responsibility for matching up the pairs of start/finish messages; when the outermost scope has finished, it writes out the information to an XML file.

As you've probably realised, the most common and obvious place to define a scope is at the entry and exit of a method, so that the nested scopes represent methods within the call stack. However a scope could be either more fine-grained or more coarse-grained than that, either with multiple scopes can be started and finished within a single method, or conversely scope could be defined at the beginning of each major subsystem within the system. The second usage (coarse-grained) is more commonplace because ideally the level of nesting should not exceed 10 levels or so, in the order of the number of major subsystems or components that interact in order to produce the response to the user. For the original system we used Spring to configure the system; we found that Spring components are a reasonable level of granularity.

The other major responsibility of the appender is to manage multiple scopes at the same time. When used in a server-side context (eg within an application server) there will be multiple concurrent threads each being invoked (typically) by different end-users. Each of these users' nested scopes must be kept separate from each other. This responsibility does in turn put an additional constraint on the format of the logging messages: it must be possible to identify their owner. This information is easy enough to make available, eg in a thread-static or from the OS itself.

Although the scoping appender uses start/finish events to define the scopes within the call stack (and thus the format of the XML file), it (being a log4j appender) also receives regular logging events. Provided that these have been written such that their ownership can be established, these messages are interleaved within the call stack. In other words, the callstack provides a context for logging messages.

2. ProfDog Browser

In order to analyze the XML file an Eclipse RCP application was developed: this is the ProfDog browser. This allows the XML files to be opened so that the timings of the different scopes within the callstacks can be compared.

Since the callstack also has embedded (regular) logging messages, these too can browsed. In practice we've found this to be more useful than using Chainsaw as a way of browsing log information.

3. Lightweight Appender

If you read the motivation section you may be wondering how we run the scoping (log4j) appender under IIS. You might also be thinking that the processing performed by this appender runs counter to our original intent of developing a very lightweight monitoring capability. And you would be right.

In fact, ProfDog has an additional appender - the lightweight appender. The role of this appender is to simply format the messages and send them via UDP to a destination port. Moreover, ProfDog has both a log4j and a log4net implementation of this appender.

Both log4j and log4net do in fact have UDP appenders. The purpose of our lightweight appender is to provide plug-in points to (a) filter out messages that don't match the appropriate format, and (b) specify how matching log messages are marshalled across the wire.

4. ProfDog Collator (part of the Browser)

If the lightweight appenders are spitting out logging events over UDP, there needs to be something listening to these messages. This is the job of the ProfDog collator.

The collator opens up a UDP socket and accepts and unmarshalls logging messages. These are then buffered for a small period of time to allow for any UDP messages that may have arrived out-of-turn. The collator then uses the ScopingAppender to actually write out the XML files representing the callstacks of separate interactions.

Originally the collator was intended to run as a stand-alone component, but it was subsequently found more convenient to just roll its functionality into the Browser. Using the Browser's GUI multiple sessions can be collated at the same time. Additionally the browser shows the state of each scope, with running totals of its depth, the number of messages, and so forth.