Introduction to Synoptic

1 Introduction

Synoptic is a tool that summarizes log files. More exactly, Synoptic takes a set of log files, and some rules that tell it how to interpret lines in those logs, and outputs a summary that concisely and accurately captures the important properties of lines in the logs. Synoptic summaries are directed graphs. Synoptic is especially intended for use with distributed systems logs, which is one of the reason we would like for you apply it to logs of systems you build for this class. This document describes features of the 490H framework that integrate with Synoptic. It is assumed that you are using framework version `v0.2` (you can test for this by passing it the `-v` option).

2 Installation

1. Synoptic depends on a tool called `dot`, which you can install as part of the Graphviz suite: `http://www.graphviz.org/`. This writeup assumes that the `dot` command is installed in `/usr/bin/dot`.

2. As of this writing, the latest release of Synoptic is version 0.0.3. You can download it (`synoptic-0.0.3.tar.gz`) here:
   `http://code.google.com/p/synoptic/downloads/list`

3. You also need to download a supporting set of files specific to 490H. This includes files containing Synoptic command line arguments for processing logs generated by the 490H framework code, as well as some sample logs that will be used as examples in this writeup. You can download the archive with these files (`490h-0.0.3.tar.gz`) from the above link.

4. Unpack the release archive into some directory. This writeup assumes that that you've unpacked things into `~/synoptic/`.

5. Unpack the supporting files archive into `~/synoptic/`.

6. The Synoptic release you unpacked is a set of jars. Synoptic is written in Java and its current interface is the command line. From within `~/synoptic/`, execute the following command to show the help screen, as a test of your installation:
   ```
   java -jar synoptic.jar -h
   ```

7. As a final test, from the same location as above, run the unit tests distributed with Synoptic:
   ```
   java -jar synoptic.jar --runTests
   ```

For more information, see the installation instructions on the website:
`http://code.google.com/p/synoptic/wiki/DocsInstallation`

3 Usage through examples

Lets first run synoptic on sample logs that you have downloaded. For these examples to work you have to (1) create an output directory – ~`/synoptic/output/`– to store Synoptic output files; and (2) make sure that you run the following commands from ~`/synoptic/`.

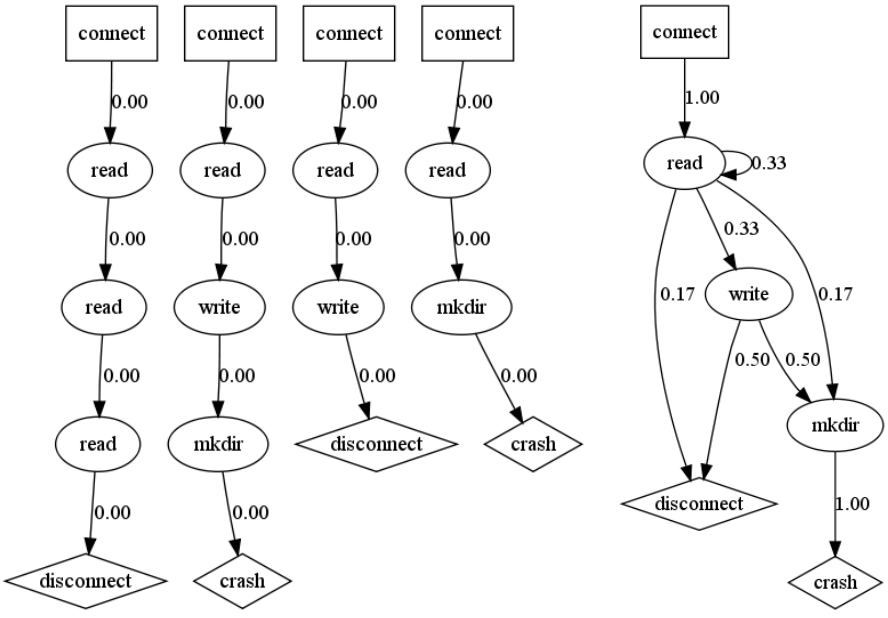
3.1 Example 1: basic usage

The following command will process `example1.log` by using command line arguments stored in `example1.args` and use the dot command at `/usr/bin/dot` for output:

```
java -jar synoptic.jar -c example1.args -d /usr/bin/dot example1.log
```

By default this command will generate four files in `output/`:

1. `example1.initial.dot`: the initial graph parsed from `example1.log` file in `dot` command file format.
2. `example1.initial.dot.png` : the graphic corresponding to `example1.initial.dot` (Figure 1(a)).
3. `example1.dot` : Synoptic's final graph in `dot` command file format.
4. `example1.dot.png` : the graphic corresponding to `example1.dot` (Figure 1(b)).



(a) example1.initial.dot.png (b) example1.dot.png

Figure 1: Example1 output. ((a)) Initial graph, ((b)) Final graph.

Figure 1 above illustrates the two outputs. The graph on the left represents the **input graph** to Synoptic. The graph on the right is the **final graph** that summarizes the input log.

The input graph is a translation of the log into a graph form. It allows you to check visually whether Synoptic correctly parsed the input. For example, `example1.log` has a trace that goes: `connect, read, read, read, disconnect`. This trace is shown as the first from the left in the input graph. You can suppress the output of the initial graph with the `--dumpInitialGraph=false` option.

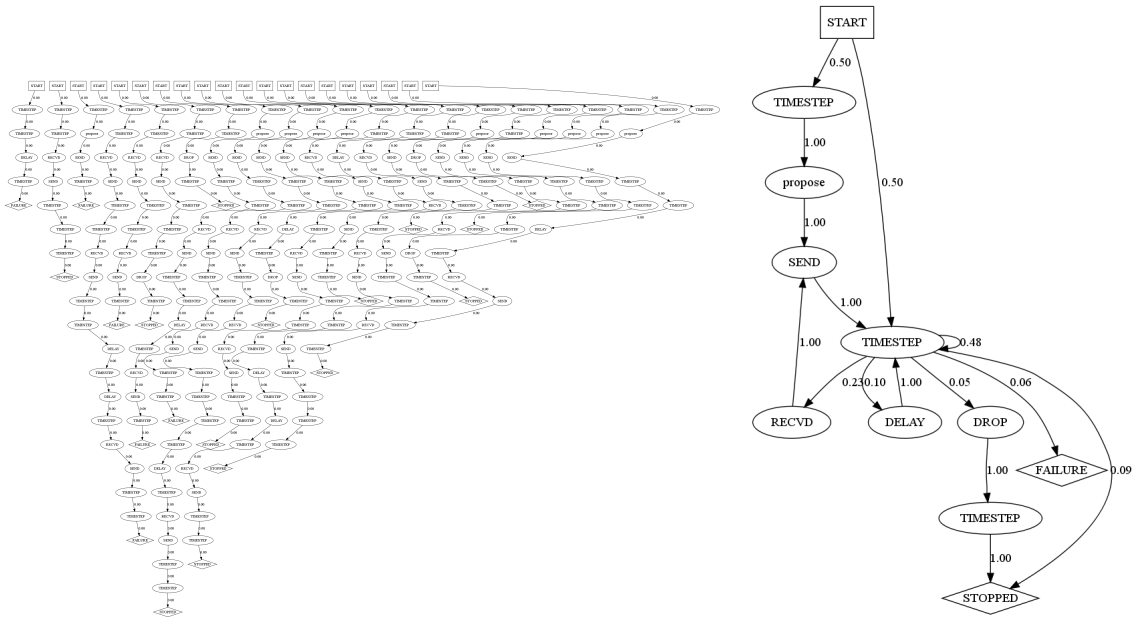
Note that the edges in these graphs are labeled with numbers, which represent transition probabilities. For the input graph, these labels are always 0.00. For the final graph a transition probability indicates the percentage of time that the transition (edge) appeared in the log file (along the paths that get to this node).

Also, in both graphs nodes that represent events or event types that are the first in a trace are represented as square nodes. Final events in a trace are represented as diamonds. All other nodes are ovals.

3.2 Example 2: TwoGenerals simulator log

The TwoGenerals problem discussed in class is implemented and released as an example in the `proj/` directory of the framework. Synoptic output from 10 executions of this example of sample synoptic logs is in the files `example2.0.log, ... , example2.9.log`. The corresponding command line arguments are in `example2.args`. Run the following command to generate the graphs below:

```
java -jar synoptic.jar -c example2.args -d /usr/bin/dot example2.*.log
```



(a) example2.initial.dot.png (b) example2.dot.png

Figure 2: Example2 output. ((a)) Initial graph, ((b)) Final graph.

3.3 Example 3: ignoring certain event types

The framework logs all possible events that might be interesting to you. The `490h.args` file contains the regular expressions to parse all of these different event types.

However, you might find yourself tracking down a bug that only implicates a subset of events. For example, a bug that manifests itself only when you use persistent storage. You can constraint Synoptic to consider just the events you are interested in by *ignoring* log lines that do not match the specified regular expressions. You can do this by (1) commenting out regular expressions for the events types that do not interest you in the `490h.args` file, and (2) passing the `-i` option to Synoptic to ignore unmatched lines.

Consider the logs from example 2. These log contains `TIMESTEP` events, which are generated by the framework whenever internal framework time is incremented. This event might not be of interest, therefore you can use `example3.args` to ignore these events (it specifies the `-i` option) by re-running on the same set of logs:

```
java -jar synoptic.jar -c example3.args -d /usr/bin/dot example2.*.log
```

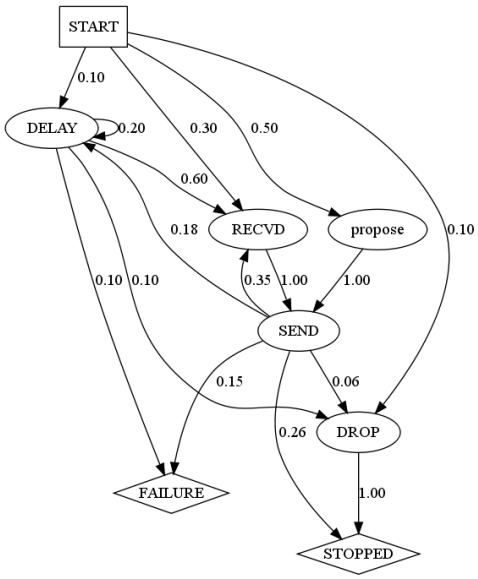


Figure 3: Example3 final graph output.

4 Logging framework events

For your convenience the 490H framework is capable of generating logs in a format that can be then parsed by synoptic. The Synoptic arguments for parsing all event types generated by the framework are in the `490h.args` file.

In fact, the framework can generate two kinds of event logs – a totally ordered log, and a partially ordered log (each event is annotated with a vector clock value). The totally ordered log can only be generated in simulator mode (with the `-s` option), while the partially ordered log can be generated in both simulator and emulator (`-e`) modes. [1] For the framework to output these logs

[1] The totally ordered log is possible in simulator mode because the simulator is implemented as a single thread.

you have to pass it appropriate command line options:

- The `-L` option takes a filename as argument for storing a **totally** ordered log of events.
- The `-l` option takes a filename as argument for storing a **partially** ordered log of events.

For example, to have the `RIOTester` example output both kinds of logs in simulator mode, you can run the following:

```
./execute.pl -L total.log -l partial.log -s -n RIOTester -c scripts/RIOTest -f 0
```

As a result of the above, `total.log` and `partial.log` files will contain the totally and partially ordered simulator event logs respectively.

4.1 Framework event types

The framework logs different kinds of event. The above examples illustrated most of these, but here is the full list:

- Packet sends and receives : `SEND, RECVD`
- Packet delays and drops : `DELAY, DROP`
- Timeouts : `TIMEOUT`
- Script file commands : `COMMAND`
- Persistent storage events : `READ, WRITE`
- Node start and stopped events : `START, STOPPED`
- Simulator time ticks : `TIMESTEP`
- User defined events : `USER-EVENT`

Each of these events types is parameterized with values. In some cases there is a set format for the event type. For example, the packet `SEND` event log line specifies the source and destination nodes, the protocol id, and the packet payload. Here is an example:

```
31 SEND src:1 dest:3 proto:0 rio-proto:20 rio-seqNum:0 rio-payload:0
```

In other cases the event log line may vary depending on the function invoked. For example, the persistent storage `WRITE` event log line has many types of formats in which the line may include or exclude the offset, and len fields depending on which method was invoked. Example log lines:

```
138 node:0 WRITE buf:'HELLO'
139 node:0 WRITE buf:'HELLO WORLD' offset:0 len:6
```

The best way to figure out the line format for an event type is to trigger the event from code and then inspect the generated log. The 490h Synoptic regular expression files illustrate how to parse most of the event types generated by the framework, but they do not always differentiate between different sub-types of the same event. For example, the two `WRITE` log lines above are treated identically.

Therefore, all events are eventually serialized during execution. The emulator, however, cannot serialize all events, therefore a totally ordered log cannot be generated in emulator mode.

4.2 User events interface

As indicated in the last example above, you can insert user defined events into Synoptic logs. To do so, use the `logSynopticEvent` method, defined in the `Node` class. You can invoke this method directly on the class you used to extend the `Node` class. This method call takes a single event string description argument which will appear last on the generated log line.

4.3 Correctly logging packet payload

The framework has no control over how your code converts Packets into byte arrays (i.e. `byte[]`). However, to log events that are parameterized with packet payloads the framework needs to know how to convert these byte arrays into strings. For this, the framework relies on the `packetBytesToString` method in the `Node` class. By default, this method uses the `Utility.byteArrayToString` method to convert Packets to strings. However, if you translate your packets into byte arrays differently, or if you would like to elide packet payload from the logs altogether, you should override this method in your derived `Node` class.

4.4 Usage Tips

- When debugging log parsing, use the `--debugParse` option. This option will print each log line and the corresponding reg-exp groups that were parsed from that line.
- When using Synoptic your first goal should always be to generate the right looking initial graph. Therefore, always look at the initial graph output. This graph will conform to how Synoptic interpreted your regular expressions.
- Synoptic produces most useful output for input logs that contain many runs of a system because it excels at compressing shared event sequences from multiple runs. Passing it a log from a single run of the system, or one in which each event type appears just once, will result in a large final graph.
- Synoptic may seem to hang or take a very long time on large logs, or on logs that have too many event types. This is a known issue. Try running it on a sub-sample of the logs, or reducing the number of event types. If the problem persists, contact us.
- Synoptic is a research project that is under active development. If you run into a problem that is not answered by this writeup and is not covered in Synoptic's online documentation, please contact us!

5 Other resources

If you run into problems or have questions about installing/using Synoptic or interpreting its output, do not hesitate to seek help. The Synoptic website contains a tutorial and additional usage information that you might find helpful. The website contains the most up to date information about the tool:
`http://code.google.com/p/synoptic/`

Synoptic is developed by many people, almost all of whom are at UW CSE. Your main point of contact with this group is Ivan: `ivan@cs.washington.edu`.