Flows and Traces

Contents

1 Workers

A **topology** is a directed graph of workers that can process messages, maintain and update state, and send the resulting messages to other workers. The following defines a worker type:

$$\begin{aligned} WT ::= & \; Msg \to State \to (State, WT \to \overline{Msg}) \\ & \times State \\ & \times StateIndex \end{aligned}$$

The WT type definition says that a **worker** is a state machine that transitions based on the message received and the current state (technically, its alphabet is the set of messages Msg). When a worker undergoes a state transition (including the identity transition), it sends to each of a defined subset of WT a sequence of messages, and, if it was not the identity transition, increments its state index.

It is worth mentioning that in this context, a "message" is simply data received at or sent by a worker. We have called this a **handoff** in other contexts to distinguish it from a **logical message**. A logical message is defined as data associated with a UUID that is processed in steps (and possibly in parallel) in a topology. In what follows, we reserve the term "message" for logical messages.

2 Flows

When designing an application, we create constraints on the way messages can flow through our topology. We can express the constraints on the paths a logical message can follow as the following recursively defined type:

$$
\begin{aligned}
Flow ::= \; & WT \\
& \mid Flow \text{ “;” } Flow \\
& \mid Flow \text{ “||” } Flow \\
& \mid \overline{Flow} \\
& \mid Flow \text{ “|” } Flow \\
& \mid \overline{Flow \text{ “|” } \ldots \text{ “|” } Flow} \text{ “(” } x \text{ “,” } y \text{ “)”}
\end{aligned}
$$

Let's look at each line of this type definition in turn. First, a $Flow$ can consist of a single worker type. This is the simplest case where a message is processed by a single worker. We use capital letters to represent worker types in a flow. The following describes a valid $Flow$ consisting of a single worker type Z:

Z

Of course, in most cases, our application will require processing by more than one worker. The simplest multi-worker $Flow$ involves a linear pipeline

of worker types. The **sequence operator** ";" represents a causal ordering between two workers (or flows). If our flow looks like

```
A -> B -> C
```

we can express it as

$$A; B; C.$$

Flows do not have to be linear. Take the following topology:

```
   ->B
  /
A-
  \
   ->C
```

Here our flow splits after A. We can represent this using the **fork operator** "||":

$$A; B||C$$

Now consider a topology where the forked paths join again at a worker D:

```
   ->B-
  /    \
A-      ->D
  \    /
   ->C-
```

We represent the **join operation** over a fork by an overline:

$$A; \overline{B||C}; D$$

A fork indicates that data must flow along all the specified paths. However, some flows might also include mutually exclusive choices. This is where the **choice operator** "|" comes in. If in the preceding example data must either flow along B to D or C to D but not both, then we would express our flow as follows:

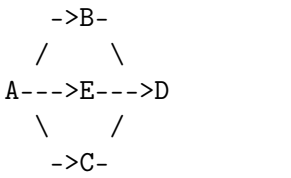
$$A; \overline{B|C}; D$$

In this case, the overline does not technically indicate a join, but is simply used to make it clearer that both B and C lead to D. [Is this how we want to do it?]

Finally, we might want to say that a flow requires something in between a fork and a choice. Consider the case where we want to say that data must either flow to D along B and C or along B or along C. What we are saying here is that data must flow along at least 1 and at most 2 of the choices. We would express this as:

$$A; \overline{B|C}(1,2); D$$

As another example of this **inclusive choice**, consider the following graph:

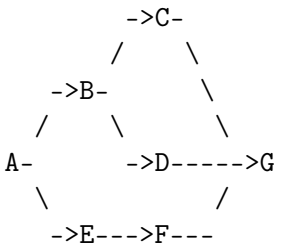


Say our specification is that data must flow along exactly two of B, E, and C to D. We would express this as

$$A; \overline{B|C|E}(2,2); D$$

Note that we could think of $A|B$ as shorthand for $\overline{A|B}(1,1)$.

Keep in mind that the binary operators described in this section can connect any two *Flows*, not just any two worker types. For example, consider this slightly more complex topology:

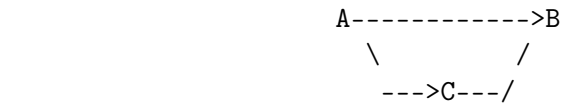


This more complex fork and join can be represented as

$$A; \overline{[B; \overline{C||D}]|||[E; F]}; G$$

Note the use of square brackets to group the conjuncts of the initial fork.

This model for expressing flows cannot capture every case as it stands. Consider the following:



How do you write this as a $Flow$? $A; B||C$ doesn't work because it fails to capture the join between A and C.

We could write something like

$$A; \overline{A||C}; B$$

but this is misleading. Data is only processed at A once, but this formula seems to say that it is processed twice at A.

A cleaner approach is to use a symbol that refers back to the origin, such as:

$$A; \overline{@||C}; B$$

Here "@" refers to A. It also implies both that processing only happens at A once and that A is joined with C at B. See the Relay Workers section below for a more in-depth explanation of the semantics of "@".

Finally, a **subflow** of a $Flow$ is any component $Flow$. For example,

$$E; F$$

is a subflow of

$$\overline{[B; \overline{C||D}]|||[E; F]}$$

2.1 ASCII Version

The ASCII version of Flow Notation is the same except that it uses [] instead of overlines for joins. It also uses [] to demarcate groupings. For example, consider a more complex example from the last section:

$$A; \overline{[B; \overline{C||D}]|||[E; F]}; G$$

The ASCII version would look like

$$A; [[B; [C||D]]|||[E; F]]; G$$

2.2 Relay Workers

"@" stands for what we can call the "Relay Worker", which is a stateless passthrough worker. Wherever X;Y shows up in a flow, you can interpolate a "@" without changing the spec. So X;Y can be converted to X;@;Y or X;@;@;Y, etc. Furthermore, wherever "@" is preceded or followed directly by a sequence operator, that "@" can be removed without changing the spec. For example, X;@;Y can be converted to X;Y, Z;@ can be converted to Z, and @;Z can be converted to Z.

"@" cannot be added or removed from a fork or choice without changing the spec, however, as it denotes the implicit expansion that occurs when a worker is joined with one of its descendents. So

$$A; \overline{@||B}; C$$

is not equivalent to A;B;C, since the latter leaves out the join at C.

To illustrate:

```
A--------->C
 \        /
  -->B-->
```

can be (and must be notationwise) thought of as:

```
A--->@---->C
 \        /
  -->B-->
```

2.3 Restrictions

A *Flow* cannot begin with a fork. This is because any traversal of a flow should be associated with a single UUID (as explained below). However, if a *Flow* began with a fork, there would be no reliable way to generate the same UUID at the forked workers.

The same reasoning implies that an inclusive choice is also an invalid starting point for a *Flow*, since an inclusive choice indicates the possibility of more than one worker processing in parallel.

An exclusive choice between two worker types, on the other hand, is a valid starting point since only one of the options is allowed in any given traversal.

Finally, a Flow cannot begin with "@", since a stateless worker cannot generate UUIDs.

3 Traversals

A *Flow* defines a spec for all the valid ways data can flow through a topology. An actual **traversal** of the topology can be represented as a *Flow* that contains no choices. This is because if a spec says you can only take one of two branches, then any actual traversal satisfying the spec will include one of those branches and not the other. On this basis, we can define the *Traversal* type as follows:

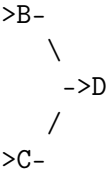
$$\begin{aligned} \mathit{Traversal} ::= {} & \mathit{WT} \\ & \mid \mathit{Traversal}\ \text{“;”}\ \mathit{Traversal} \\ & \mid \mathit{Traversal}\ \text{“||”}\ \mathit{Traversal} \\ & \mid \overline{\mathit{Traversal}} \end{aligned}$$

Notice that this is the same type definition provided for *Flow* except without choice or inclusive choice.

For example, say we have the following *Flow*:

$$A; B|C; D$$

corresponding to the following topology:



$B|C$ says that any logical message must be processed along B or C, but not both. Each logical message that is actually processed will thus correspond to one of two possible Traversals:

$$A; B; D$$

or

$$A; C; D$$

4 Traces

In order to check whether a traversal of a topology satisfies a given spec (both in terms of the flow and other properties like latency), we are going to have individual workers send reports to what we're calling a **Trace Receiver** whenever they process a logical message. We will uniquely identify a logical message using a UUID. We discussed having workers send a **trace report** matching the following type:

$$\begin{aligned} TraceReport ::= & \; UUID \\ & \times WT \\ & \times InstanceID \\ & \times StateIndex \\ & \times UnderivedState \\ & \times Hop \\ & \times Hash \end{aligned}$$

A **trace** corresponds to a set of $TraceReports$ bearing a single UUID. We can write its type as

$$Trace ::= (\text{UUID}, (\text{TraceReport}, \text{Stamp}))$$

The reporting worker identifies itself and its state by reporting its type, its instance id, its state index, and any underived state that contributed to the result of its processing and/or its state transition. Furthermore, in order to help in reconstructing the entire traversal, we decided to have it send the current hop count associated with the UUID and a hash value meant to disambiguate between identical hop counts found along parallel processing paths.

The hash is constructed by feeding a hash function the local InstanceID and hop count, the result of which is then added to the hashes of all the hashing worker's predecessors' hashes. This means that each worker must forward its hash value downstream so that its immediate successors can use that hash in calculating their own hash values.

In order to determine the actual predecessors of a $TraceReport$, you would need to try the possible predecessor hashes. Given a $Flow$, you can find the valid predecessors of a WT. First find every appearance of that WT in

the $Flow$ either preceded directly by ";" or as a subflow of a $Flow$ preceded directly by a ";". Any ";" will be preceded by either a WT, a join, or a choice. Every WT that appears either directly in front of the ";" or as a conjunct in a join or as a disjunct in a choice preceding the ";" is a valid predecessor. You can use this list of valid predecessors to try possibilities when testing a hash.

However, this type information is not enough. We need to be able to identify the actual instances of the valid types, since the possible predecessor hashes are constructed from an instance id and a hop count. It doesn't seem likely the hash approach will avoid combinatorial explosion. For example, say a worker has 10 possible predecessor instances (this is a conservative example, since there's nothing to prevent a topology from including a large number of workers). Even if we know that a given $TraceReport$ has 4 actual predecessors (perhaps by adding a predecessor count to the $TraceReport$ type), that leaves us with

$$\binom{10}{4}$$

possibilities, which is 210 sets of 4 predecessors. If we had 100 possible predecessors, we'd have

$$\binom{100}{4}$$

or 3,921,225 possibilities. This indicates that the algorithm doesn't scale.

For this reason, the new approach is to have a worker send its predecessor hashes as well as its own hash. These hashes are calculated without needing a hop count. The predecessor hashes can be used to uniquely identify predecessor $TraceReports$ within a $Trace$ without having to try out possibilities.

According to this approach, we would define a $TraceReport$ as follows:

$$\begin{aligned} TraceReport ::= & \; UUID \\ & \times WT \\ & \times InstanceID \\ & \times StateIndex \\ & \times UnderivedState \\ & \times PredecessorHashes \end{aligned}$$

The trade-off is that a *TraceReport* would require more memory, growing with the number of actual predecessors for the relevant traversal.

4.1 Trace Uniqueness

The first property we've identified of our system is the following:

Trace Uniqueness: A single trace corresponds to exactly one traversal.

4.2 Deriving a Traversal from a Trace

Algorithm for deriving a *Traversal* from a *Trace* (provided we use the alternative to hashing outlined above):

1. Build a predecessor graph with edges directed from a node to its predecessors (uniquely identified by their trace report hashes). The sink of this graph is the source of the traversal. The rule against beginning a *Flow* with a fork or inclusive choice entails that there will be only one source per traversal.
2. Reverse the graph to get directed edges going in the direction of the traversal.
3. Beginning from the source discovered in step 1, build the *Traversal*, using forks for multiple edges extending from one node and joins for edges joining at a node. Though the graph uses worker instances as nodes, the resulting *Traversal* will denote these nodes by their type only.

5 Replaying

The following formula still needs work: \
traces(A_i) ::= $\{\mathrm{t} \mid \mathrm{t} \in \mathrm{Trace} \wedge \exists\ \mathrm{n} \leq \mathrm{i}.\mathrm{A}_\mathrm{n} \in \mathrm{t}\}$

[We should discuss this as a group.]