Distributed DDlog Reconfiguration Examples

Consider the D3log program from our previous note ([1]) shown in Figure 1 and Listings 1 and 2:

Listing 1: Rules for switch $S1$

```
input relation host(hostID: int, switchID: int)
input relation S3.blacklist(hostID: int)

output relation S1.host(hostID: int)
output relation S1.blacklist(hostID: int)

S1.host(id) :- host(id, 1).
S1.blacklist(id) :- S3.blacklist(id, 1).
```

Listing 2: Rules for switch $S3$

```
input relation S1.host(hostID: int)
input relation S2.host(hostID: int)
input relation blacklist(hostID: int)

output relation S3.host(hostID: int, switchID: int)
output relation S3.blacklist(hostID: int,
        switchID: int)

S3.host(id, 1) :- S1.host(id).
S3.host(id, 2) :- S2.host(id).

S3.blacklist(hostID, switchID) :- blacklist(hostID),
                                  S3.host(hostID, switchID).
```

We describe a reconfiguration scenario where this program recovers from a central switch (`S3`) failure, preserving eventually consistent input/output behavior while avoiding complete recomputation. To do so we model the application as a collection of *incremental processes* and channels (Figure 2). Each process takes a stream of updates to 0 or more input relations and outputs a stream of updates to 0 or more output relations that are sent via channels to other

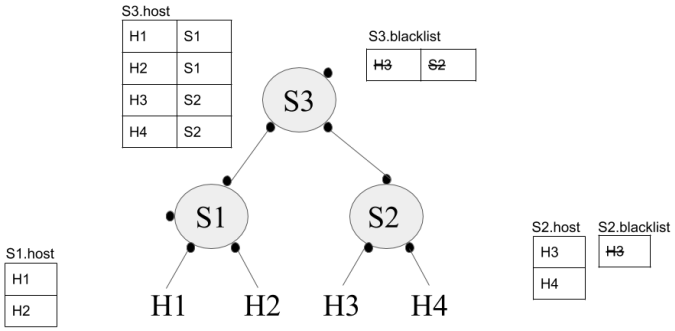

Figure 1:

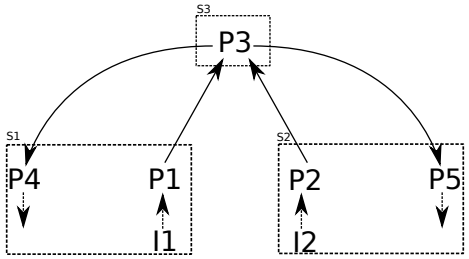

Figure 2:

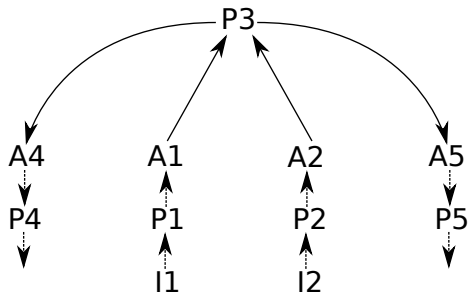

Figure 3:

processes. We use solid lines for distributed channels and dashed lines for local channels.

This notation abstracts away specific input and output relations and rules that each process represents. For example, process `P1` represents local computation in `S1` that receives updates to the `host` relation from input source `I1` and sends updates to `S1.host` via a channel to the central switch. Process `P4` receives updates to the `S3.blacklist` relation from `S3` and outputs `S1.blacklist` to a local sink (not shown in the diagram). Process `P3` implements the entire central switch logic.

In order to enable failure recovery we attach special *accumulator* processes, `A1`, `A2`, `A4`, and `A5`, to inputs and outputs of processes `P1`, `P2`, `P4` and `P5` respectively, as shown in Figure 3:

During normal operation, an accumulator behaves as a pass-through filter, instantly copying all input updates to its output stream. Internally it accumulates all updates, so that at any point the accumulator stores a complete snapshot of its input relation. When accumulator's *input* stream gets disconnected, e.g., due to a network failure, this has the effect of the input relation becoming empty. The accumulator then outputs a bunch of `delete` commands, one for each record in the accumulated snapshot of the input relation and clears it internal cache. For example, when the central switch fails, process `P3` and all links between `P3` and the other to switches are terminated (Figure 4). This causes accumulators `A4` and `A5` to send batches of updates to their downstream processes `P4` and `P5` to the effect of clearing their input relations:

When accumulator's *output* channel is disconnected, the accumulator keeps its cached state and continues tracking input changes. When a new downstream process is eventually attached to the accumulator, the accumulator brings it up to speed by outputting all its cached records in a single transaction.

Figure 5 shows how this capability facilitates recovery.

We create a fresh instance `P3'` of process `P3`, possibly on a different physical host and establish channels between `P3'` and other processes. As soon as channel `A1-P3'` is established, the accumulator `A1` dumps its cached state into the channel (step 1 in the diagram). As process `P3'` receives updates from `A1`

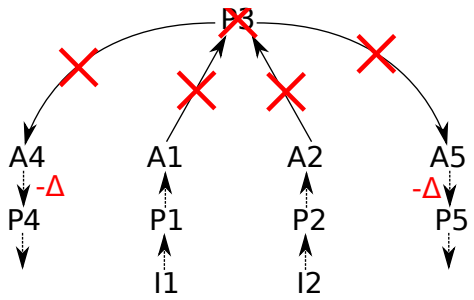

Figure 4:

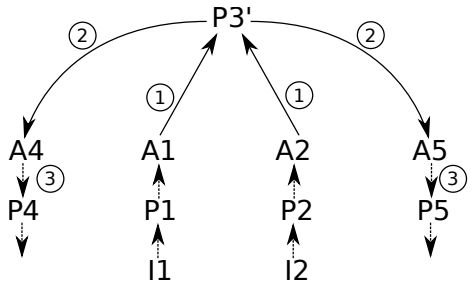

Figure 5:

and A2, it updates its outputs, which are streamed to P4 and P5 (via A4 and A5 respectively) (steps 2 and 3 in the diagram). Eventually the system gets back into a consistent state.

Note that this recovery procedure does not require centralized coordination. For instance, updates from A1 and A2 can reach P3' in any order.

As a further optimization, we can insert another gadget in the A4-P4 channel. In a typical failure recovery scenario, this channel will first see a transaction that clears the entire input relation ($-\Delta$) followed by several transactions that insert a set of records nearly identical to the deleted set. If we accumulate these updates and delay them until recovery is over (requires coordination with other nodes) or for a fixed timeout, this will dramatically reduce the amount of computation that P4 must perform, as well as the total size of updates it outputs.

References

[1] Sallinger, S. Distributed Differential Datalog.