Rescaling of Timely Dataflow

Lorenzo Selvatici

Aug 2019

1 Introduction

This document summarizes the work done to support dynamic rescaling of timely dataflow[1].

In a nutshell, timely runs a distributed dataflow computation in a cluster. The shape of the cluster is fixed when first initializing the computation: how many timely processes (likely spread across several machines) and how many worker threads per timely process. With this project, we allow the addition of new worker processes to the cluster.

In long running jobs with unpredictable workloads, it is likely that the initial configuration will not be the ideal one. As such, there is the need to scale-out by adding more workers (and scale-in, by removing worker processes).

2 Timely Model

Very briefly, we point out some of timely design choices that are relevant to the rescaling project:

- each worker has a copy of the entire dataflow: there is no concept of per-operator parallelism, each worker executes every operator and communicates with every other worker in an all-to-all communication pattern.
- asynchronous computation: there is no master worker that coordinates and schedules the workers for execution, this implies that is also not easy to get a consistent view of the state of the system without stalling the computation.
- the progress tracking is at the core of timely computational model, new worker should make sure they can contribute to the progress tracking protocol without violating its safety properties.

[1] https://github.com/LorenzSelv/timely-dataflow/tree/rescaling-p2p

3 Rescaling the computation

We now go into details of how the rescaling is actually implemented. Firstly, a high-level overview of the key points that need to be dealt with is given. Secondly, we present the design options for the initialization of the progress tracking state for the new worker. Lastly, we describe the integration with Megaphone.

3.1 Communication Infrastructure

Rescaling the computation is possible only when running in cluster mode: multiple timely processes, each wrapping several worker threads, have established connections to each other and exchange both data and progress updates.

In this setting, "adding more workers" means adding another timely process with the same number of worker threads of every other timely process in the cluster.

Communication among workers happens in two instances:

- `exchange` operator
- progress updates broadcast

In both cases, communication is enabled by *allocating a channel* which contains an endpoint to every other worker for both sending and receiving (`Pusher` and `Puller` traits, respectively).

Channel allocations happen while building the dataflow within the `worker::dataflow` function call.

If a new worker process joins the cluster, we need to *back-fill* these previously allocated channels, so that they have an endpoint to the new worker as well as the previous ones. We do this by storing a closure for each past allocation, so that when the new worker initiates the connection we can invoke these closures which add the supplied pushers to the list of pushers forming the channel.

Each channel is associated with a specific data type. Thus, we cannot simply store a map of `(channel id => channel handle)`, as collections can be generic but also need to be homogeneous. One might work around this by using trait objects, but then the channel would be associated with the trait object itself rather than the concrete type.

Moreover, fast-forwarding a bit, having a closure turned out to be very handy when implementing the bootstrap protocol to initialize the new worker progress tracker: when adding the new pusher to the list, we also push a *bootstrap message* that informs the new worker about the next progress-update sequence number it will receive from that direct connection.

When running in cluster mode, each worker process spawns an extra-thread that will accept incoming connections from workers joining the cluster. Upon accepting the connection, the thread will spawn a pair of `send` and `recv` network threads, perform

some bookkeeping and finally inform the worker threads about the new timely process that is trying to join the cluster. This is achieved by sending a `RescaleMessage` on a shared `mpsc` channel.

Worker threads need to explicitly check for rescale messages via the `worker::rescale` function call. This is done in the very beginning of the `worker::step` function.

With the exception of the *bootstrap* protocol that we will talk about in a later section, this is all from the perspective of worker processes already in the cluster.

3.2 New Worker Initialization

Initialization of the new worker boils down to two things:

- building the dataflow (possibly more than one)
- initializing the progress tracker

We will now talk about them in detail.

3.2.1 Building the dataflow

Since the program we use to run the new process is the same to the one of other processes, the construction of the dataflow comes for free. The single and very important difference is how we handle *capabilities*. In particular, generic operators such as `source` and `unary_frontier` supply a capability to the user-provided constructor, which can use it request notifications or store it to retain the ability of producing output at a later time.

Clearly, as the new worker can join at any time in the computation, we must not supply capabilities for timestamps (epochs) that have been closed already as that would allow the worker to produce output "in the past".

Ideally, one would like to have a capability consistent with the frontier of that operator at the time of joining the cluster: if the new worker joins when an operator happens to be at some timestamp `t`, it would be fine to provide a capability for such timestamp to that operator.

Unfortunately, there are some complications in trying to do this:

- As already mentioned, capabilities are passed to the constructor of the generic operators while building the dataflow. However, at least with the current implementation of the bootstrap protocol, we need to build the dataflow *before* actually performing the protocol. But then we do not know the operator frontier and thus also the right capability to supply to the constructor.
- If we do supply a capability for a certain timestamp `t`, we need to ensure that no other worker will ever consider that timestamp "closed" before the new worker discard the capability for that timestamp (e.g. by dropping or downgrading it).

For the first item, the issue could be worked-around by sending a map to the new worker which, for each operator, specifies the frontier of that operator. Then, while building the dataflow, the new worker would look up in the map the frontiers and supply appropriate capabilities.

An alternative solution, and the one currently implemented, is to *not* supply capabilities at all to the new worker. This approach does bring some limitations:

- Operators can produce output only in response to some input (which comes with the associated capability for that timestamp). As such, `source` operators are useless for new workers: they do not hold any capability and cannot produce any output.
- Operators cannot request notifications for future times unless they use capabilities that came with input data.

There is an asymmetry between workers which hold capabilities (initial workers present in the cluster) and are capable of injecting new data in the dataflow (via the `source` operators) and workers that do not hold capabilities as they joined the cluster at a later time. So, while we can add worker at runtime, there are some "special workers". Looking from a fault-tolerant perspective, where we would like to allow arbitrary worker to crash without compromising the execution, this might be a potential showstopper.

For the second item, the bootstrap worker (see later section about the bootstrapping protocol) should emit a `(t, +1)`[2] pointstamp (progress update), to ensure that the timestamp `t` will not be closed until the corresponding `(t, -1)` pointstamp is emitted. Such `(t, -1)` will be emitted by the new worker upon downgrading the capability.

3.2.2 Initializing the Progress Tracker

The progress tracking protocol is arguably the most important component at the core of timely computational model. As such we need to ensure that new workers have an up-to-date and correct view of the progress state.

When building the dataflow, each operator is supplied by default with capabilities for timestamp `0` (or the default analogous for different timestamp types). Some operators do not require capabilities and simply discard them. Other operators use them to request notification, produce output, etc. These operations on capabilities are associated with the emission of progress updates, made of a pair `(pointstamp, delta)`. These progress updates are broadcasted by every worker to all other workers via the established TCP connections between pair of processes.

There are two alternatives to initialize the progress state of the new worker:

- Reconstruct a consistent view by all the frontiers of the operators, which have changed over time as a consequence of progress updates.

[2] it should actually be `num_worker_threads` instead of `+1`, as every new worker thread should get a capability

- Record and accumulate every progress update that has been sent so that we can apply them again.

We implemented the second option, as it is easier to reason about and hard to get wrong. Also, as progress updates tend to cancel each other out (`+1` and `-1` pairs), we *expect* such accumulation to remain fairly small, no matter how long the computation has been run for. We should add proper instrumentation to the code to verify this claim.

In the next two sections we present the two alternatives for implementing the above idea: `pubsub`-based approach and `peer-to-peer`-based approach, both of them have been implemented to some extent, but `peer-to-peer` turned out to be a better option.

3.3 Pub-Sub Approach

One possible design would be to swap the all-to-all communication pattern for progress updates with an external `pubsub` system. Such system would have established connection to all workers. Each worker would send its progress updates that will be broadcasted on its behalf. Each worker then reads progress updates from every other worker.

Progress updates from each worker could be appended to a queue and when every other worker currently in the cluster has read the progress updates up to a certain point, updates before that point can be safely compacted (most term would cancel out).

When a new worker joins the cluster, it would "subscribe" to every other worker's queue, initialize its progress tracking state using the compacted updates and finally start reading new updates from the append-only queue of updates.

A possible implementation of such queue mechanism has been implemented[3].

3.3.1 Discussion

While this sounds like a clean and simple idea, there are a few complications:

- To perform updates compaction, the `pubsub` system needs to be aware of timestamp data types in the dataflow. Since these timestamps depend on the dataflow that has been constructed, the `pubsub` system needs to build the same dataflow, unless we implement some weird export mechanism. Moreover, the `pubsub` system needs to use timely data types, which would require a circular dependency between the two otherwise-unrelated crates.

- More on a philosophical note, adding this external system, which also represents a single point of failure, goes a bit against the peer-to-peer spirit of timely.

[3] `https://github.com/LorenzSelv/pubsub`

From performances point of view, there would be definitely some overhead associated with the `pubsub` system itself. Among other things, now there are two hop that a message needs to go through before reaching its destination. On the other hand, one could easily optimize the amount of progress messages sent around: you could have a single subscriber per process, so that the same progress messages are not duplicated because of multiple internal worker threads[4]. This might also solve the communication volume bottleneck that some experiments show to prevent timely from scaling-out after a certain limit[5].

3.4 Peer-to-Peer Approach

An alternative design would be to have the new worker initializes its progress tracking state by selecting some *bootstrap server* and perform some *bootstrap protocol*.

In particular, the bootstrap server is simply another worker that is selected to help the new worker initializing its progress tracking state.

The rescale message received by the worker thread contains a flag signaling if the worker thread was selected as the bootstrap server, and in that case it would initiate the bootstrap protocol.

3.4.1 Bootstrap Protocol

Before describing the protocol, we presents some changes needed to make it possible. To begin with, the server side of the protocol is performed in the `Worker::rescale`[6] and `TcpAllocator::rescale`[7] functions; the client side of the protocol is performed in the `bootstrap_worker_client`[8] function.

Somewhat similarly to the previous approach, each `Progcaster`[9] keeps an accumulation of all progress updates as a `ChangeBatch`. We define the progress state as this accumulation plus the information about the last sequence number included in the state for each worker.

The bootstrap server, while performing the protocol, does not perform other work. Since it needs to use non-thread-safe data structures, it was not possible to spawn a separate thread to perform the protocol concurrently.

We will refer to the new worker joining the cluster as "bootstrap client". The bootstrap protocol (also depicted in figure 1) consists of the following steps:

[4] in the current implementation, each progress updates is sent to every worker thread, thus there are duplicate messages between pair of processes

[5] mostly referring to the results in the Noria paper, but we should perform some proper experiments to investigate this further

[6] `timely/src/worker.rs`

[7] `communication/src/allocator/zero_copy/allocator.rs`

[8] `communication/src/rescaling/bootstrap.rs`

[9] the entity responsible for broadcasting and receiving progress updates for a specific scope

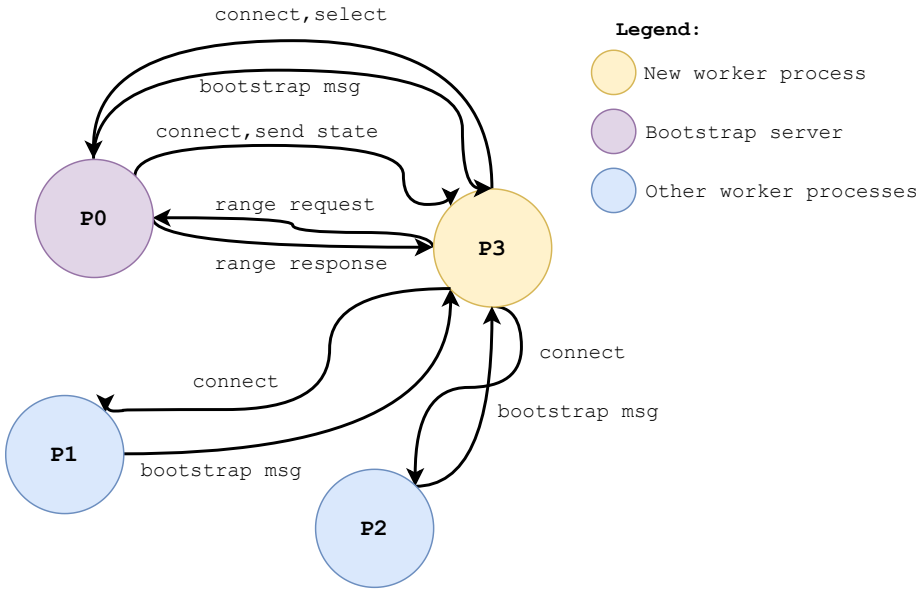

Figure 1: Bootstrap protocol, timely processes are shown as circles

1. `Bootstrap client waits for an incoming connection at some arbitrary bootstrap address.`
2. `Bootstrap server initiates the connection to that same address.`
3. `Bootstrap server sends the progress tracking state to the new worker and start recording following progress updates.` The progress tracking state has been defined above. Recording messages simply means appending them to a list (different for each worker) so that we can access them at a later time. The lists are cleared at the end of the protocol.
4. `Bootstrap server listens for incoming progress-updates-range requests.`
5. `Bootstrap client inspects direct TCP connections with other workers, where it will find the bootstrap messages, containing the next progress-update sequence number it will read from that channel.` The bootstrap message is the first message that is sent over the newly-established TCP connection.
6. `Bootstrap client computes the missing progress updates ranges by comparing the sequence numbers, in the state and the ones in the bootstrap messages.` Since the new worker has the guarantee that it will see all progress updates with sequence number greater or equal to the one in the bootstrap message, filling the missing gaps between the received progress state and the such sequence numbers guarantees that it will have seen all progress updates.
7. `Bootstrap client sends progress-updates-range requests to the bootstrap server.`

8. `Bootstrap server sends progress-updates-range responses to the bootstrap client.` A progress-updates-range request has the format `(scope_id, worker_index, start/end sequence number)` which uniquely identifies some updates range. If the bootstrap server cannot fulfill the request as it has not seen all requested messages yet, it will pull more progress updates from the channels with the other workers. Eventually, all required progress will be received also by the bootstrap server which will be then able to fulfill the request.

9. `Bootstrap client terminates the bootstrapping protocol by closing the connection.`

3.5 Megaphone Integration

After the new worker has joined the cluster and initialized its progress tracking state, it is ready to perform some actual work. If the dataflow is using plain `exchange` operators, the new worker will start receiving those input data that hash to its index (modulo the new number of peers).

This "side effect" might be acceptable for stateless operator or if the correctness of your computation does *not* rely on the fact that a worker sees all the input data that hash to the same value.

Most non-trivial operators, however, keep some state used to produce the output when combined to the input data. A simple WordCount dataflow is a perfect example: to emit the correct count for a word, a worker should see all occurrences for that word.

As a result, we need to rely on some sort of routing table that gives us such guarantee. In order to avoid unnecessary overhead for those applications that do not require rescaling or do not need such routing guarantee, we decided to not provide this feature as part of core timely: we rely on Megaphone instead.

Two main changes where needed to support rescaling operation in Megaphone, we will now describe them.

3.5.1 Variable number of peers

After a rescaling operation, the numbers of peers in the cluster has changed. Thus, we must keep a reference that reflects the current value rather than a simple integer.

3.5.2 Initializing the new worker Routing State

When a new worker joins the cluster, it initializes its dataflow and progress tracking state. However, the bootstrap protocol does not perform any initialization of operators' internal state, such as the routing table for Megaphone's stateful operators.

To ensure correctness, we need to make sure that input data are properly routed even by new workers.

As a result, we slightly modified the structure of Megaphone's F operator (the one storing the routing table and receiving re-configuration commands) by adding a

feedback connection with an `exchange` operator to transfer the routing state from the bootstrap server to the new worker. The routing state is transferred when the newly introduced `Bootstrap` command is received. The routing state is made of the currently active routing table plus all future re-configurations.

A rescaling operation consists of the following steps:

- The new timely process is spawned, it setups connections and performs the bootstrap protocol. At this point, the routing state is not initialized, thus the new worker must not process any input yet.
- The controller must send a `Boostrap` command in the control commands stream. The command payload specifies the indices of the bootstrap server and of the new worker. It must send a `Bootstrap` command for each new worker thread that joined the cluster with the new timely process.
- Upon receiving the `Bootstrap` command, the bootstrap server send its routing state to the new worker using the feedback-exchange connection we described above.
- Upon receiving the routing state, the new worker implants it and can now start to consume input data and route them appropriately.

For ease when reasoning about the correctness of the protocol that initializes the routing state, we require that no other control command is issued for the same timestamp of a `Bootstrap` command.

Moreover, we also require that there are no pending configurations changes for which corresponding control notifications have been already delivered (meaning that the input frontier of the operator reached the same totally-ordered timestamp of the control command, but the configuration change has not been applied yet). The reason for this extra constraint is the impossibility to transfer to the new worker the capability associated with the reconfiguration.

We do allow, on the other hand, pending configuration for which notifications have not been delivered yet ("future" re-configurations). Since we have the guarantee that they have timestamp strictly larger than the bootstrap time, the new worker can use the capability associated with the receival of the routing state to setup the corresponding notifications.

4 Writing Rescalable Timely Programs

4.1 Stateless Operators only

Below is pasted the code for the rescalable `HelloWorld` timely program:

```rust
extern crate timely;

use timely::dataflow::{InputHandle, ProbeHandle};
use timely::dataflow::operators::{Input, Exchange, Inspect, Probe};

fn main() {
    timely::execute_from_args(std::env::args(), |worker| {

        let index = worker.index();
        let mut input = InputHandle::new();
        let mut probe = ProbeHandle::new();

        worker.dataflow(|scope| {
            scope.input_from(&mut input)
                 .exchange(|x| *x)
                 .inspect(move |x| println!("worker {}: seen {}", index, x))
                 .probe_with(&mut probe);
        });

        // if the worker is a new worker joining the cluster,
        // perform the bootstrap protocol.
        if worker.bootstrap() { return; }

        // introduce data and watch!
        for round in 0..10 {
            if index == 0 {
                std::thread::sleep(std::time::Duration::from_secs(1));
                input.send(round);
            }
            input.advance_to(round + 1);
            while probe.less_than(input.time()) {
                worker.step();
            }
        }
    }).unwrap();
}
```

The only difference is the call to `worker.bootstrap()` highlighted in the code. It is a no-op for "normal" workers. For new workers joining the cluster, however, it will perform the bootstrapping protocol to initialize the progress tracking state of the new worker itself. It is of critical importance that the call is made right after building *all* dataflows, as it will attempt to initialize the progress state for all dataflows that other workers present in the cluster have built.

4.1.1 Sample usage and execution

Let us compile and run the above program[10]. Start a cluster with two timely processes, each with a single worker thread (run in two different terminals):

```
$ cargo run --package timely --example rescaling_hello -- -n2 -w1 -p0
$ cargo run --package timely --example rescaling_hello -- -n2 -w1 -p1
```

Worker 0 will see all even numbers, worker 1 will see all odd numbers.
Before the computation has finished, spawn a new timely process, again with a single worker:

```
$ cargo run --package timely --example rescaling_hello -- -n2 -w1 -p2 -j 0 --nn 3
```

Note that there are now two extra arguments:

- `-j 0` or `--join 0` means that the worker should join the cluster, using worker 0 as the bootstrap server. Any worker can be the bootstrap server.
- `--nn 3` means that the new number of timely processes in the cluster is now 3.

Below we paste the output of a sample execution:

```
$ cargo run --package timely --example rescaling_hello -- -n2 -w1 -p0
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/examples/rescaling_hello -n2 -w1 -p0`
worker 0: seen 0
worker 0: seen 2
worker 0: seen 4
worker 0:       connection from worker 2, bootstrap address is [Some(127.0.0.1:9002)]
bootstrap worker server done!
worker 0: seen 6
worker 0: seen 9

$ cargo run --package timely --example rescaling_hello -- -n2 -w1 -p1
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/examples/rescaling_hello -n2 -w1 -p1`
worker 1: seen 1
worker 1: seen 3
worker 1: seen 5
worker 1:       connection from worker 2, bootstrap address is [None]
worker 1: seen 7

$ cargo run --package timely --example rescaling_hello -- -n2 -w1 -p2 -j0 --nn 3
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
```

[10] source code can be found in at `timely/examples/rescaling/hello.rs`

```
     Running `target/debug/examples/rescaling_hello -n2 -w1 -p2 -j0 --nn 3`
[bootstrap client] connected to worker 0
workers left: {0, 1}
workers left: {1}
[W2] sent updates range ProgressUpdatesRange { channel_id: 8, worker_index: 1, start_s
[W2] got updates range response
[W2] applied updates range response
done bootstrapping a worker
worker 2: seen 8
```

Ignoring some debug output, we see that after worker 2 has joined the cluster, it receives the input integer 8, as `8 mod 3 = 2`. Same reasoning for worker 0 (and worker 1), it was previously seeing only even numbers (`X mod 2 = 0`), but after the rescaling operation it receives 6 and 9 as they both result in 0 after the `mod 3` operation (where `3` is the new number of peers in the cluster).

4.2 Stateful operators

Below is a simple `WordCount` example, where we use Megaphone's `stateful_state_machine` interface to guarantee correctness of emitted word counts (some parts are omitted for brevity):

```rust
fn main() {
    timely::execute_from_args(std::env::args(), |worker| {
        let mut lines_in = InputHandle::new();
        let mut control_in = InputHandle::new();
        let widx = worker.index();

        worker.dataflow::<usize, _, _>(|scope| {
            let control = control_in.to_stream(scope).broadcast();

            let words_in =
                lines_in
                    .to_stream(scope)
                    .flat_map(|text: String|
                        text.split_whitespace()
                            .map(move |word| (word.to_owned(), 1))
                            .collect::<Vec<_>>()
                    );

            words_in
                // the routing table of an arbitrary worker should route properly
                .exchange(|(word, _)| calculate_hash(word))
```

```
                .stateful_state_machine(|key: &String, val, agg: &mut u64| {
                    *agg += val;
                    (false, Some((key.clone(), *agg)))
                }, |key| calculate_hash(key), &control)
                .inspect(move |x| println!("[W{}] stateful seen: {:?}", widx, x))
        });

        if worker.bootstrap() { return; }
        if worker.index() == 0 { /* send lines and reconfiguration commands */ }
    }).unwrap()
}
```

Differently from before, when the new worker joins the cluster, it will not receive any input until some reconfiguration command moves the ownership of some of the bins to it.

A short demo is available at `https://www.youtube.com/watch?v=Zsf-eMvHUxU`.

5 Evaluation

Below are listed changes that might cause an overhead to core timely when running in cluster mode, even if the rescaling feature is not used:

- an additional thread is spawned and listens at some `ip:port` for incoming connection.
- the `allocate` function used to allocate a inter-worker communication channel now returns a list of pushers wrapped in a `Rc<RefCell<..>>` so it can be mutated when rescaling. The pointer need to be dereferenced before usage.
- Progress updates are accumulated in a `ChangeBatch` by each `Progcaster` (this is also true for non-cluster mode).

Below is a list of experiments that I believe would be useful to perform to assess performances and back-up some claims with data:

- Steady-state overhead compared to timely master: how the changes listed above affect the performances in practice.
- Plot per-tuple latency versus time (epochs): how adding a new timely process affects latency both during and after the rescaling operation has complete.
- Measure the size of the progress state over time (accumulated progress updates), as a function of number of workers and workload type (communication-heavy

with lots of exchanges versus in-operator heavy computation and minimal communication). We believe that the progress state does not grow in steady-state execution for a fixed number of workers and workload type, as most progress updates cancel out (`+1/-1` pairs).

- Breakdown of timing in the bootstrap protocol.

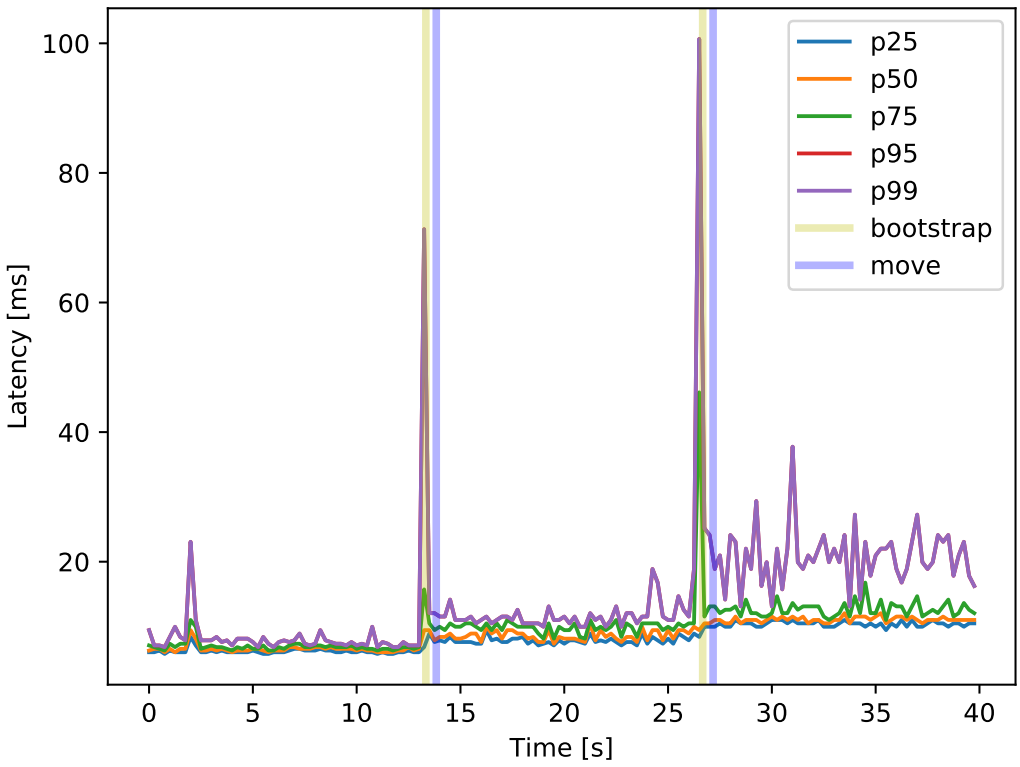

Figure 2: Latency percentiles for an initial setup of 2x2 workers and 2 rescaling operations, each adding 2 extra workers

Due to some time constraints, we did not get the chance to run many experiments. In figure 2 is depicted a (not very exciting) plot of per-tuple latencies. The experiment setup was as follows:

- everything was run on single machine (my personal laptop – Dell XPS with intel core i7 8th Gen, 16GB RAM)
- the initial cluster was made of 2 timely processes, each with 2 worker threads (thus a total of 4 workers)

- two rescaling operations were performed, each time adding an additional timely process also with 2 worker threads.

The dataflow used is a simple `WordCount`[11] program, where only a single worker samples lines and injects them into the dataflow at a rate of 100 lines per second each with 100 words in it. Lines are then distributed in a round-robin fashion among workers. The `flat_map` operator splits lines by whitespaces and emit pairs `(word, 1)`. The `stateful_state_machine` function provided by Megaphone then aggregates the elements by key and counts the occurences.

From the plot we see that the latency is pretty much constant in the beginning. In correspondence of the first rescaling operation, there is a spike in latency: a worker is busy performing the bootstrap protocol and the computation is slowed down.

After the new worker joined the cluster we move the ownership of some bins to the new worker with a Megaphone re-configuration command. However, the latency does not benefit from this rescaling operation as the latency tends to be higher. Out hypothesis is that the workers already present in the cluster were not saturated before the rescaling operation and adding more workers simply increased the communication costs (exchange data and progress messages) which outweighted the benefit of distributing the workload more evenly among the workers. The second rescaling operation presents a similar behaviour.

A better experiment would present saturation in the beginning and, after the rescaling operation a decrease in latency due to the reduced per-worker-load.

I tried to increase the rate of the input, or the length of the lines, but I decided to stop after freezing my machine twice and having to restart it.

6 Limitations and Future Work

We now highlight what are the limitations at the current stage of development.

6.1 Removal of Workers

One missing, but very important feature for a fully rescalable dataflow computation is allowing the removal of workers from the cluster. While the technical implementation details might not be hardest to deal with, it has a lot of overlapping aspects with the fault-tolerance chapter.

A worker leaving the cluster is *noticed* by the other remaining workers by TCP connections that have been dropped. At the moment, if a connection is interrupted, the computation is shut-down. Instead, one should remove all the endpoints pointing to that worker process and just keep going with the computation.

In the code there are several places where it is assumed that the worker indices are in the range `(0..peers)`, since this assumption would not hold anymore, all the

[11] `https://github.com/LorenzSelv/rescaling-examples/blob/master/src/bin/benchmark.rs`

associated data structures that rely on these indices should be turned into `HashMap`s. There are also some subtle and important things one should think about, among the others:

- If the worker leaving the cluster is holding capabilities and does not get the chance to discard them, the other workers have no way to know that those capabilities belonged to the leaving worker, and they would stall as a result.
- If the worker leaving the cluster is owning any data (e.g. is associated with some bins in the routing table of Megaphone's stateful operator) this would cause some serious problems. If we assume that we *decide* when a worker leaves the cluster, we can first transfer state and data ownership to other workers and then kill the worker.

6.2 New worker has no capabilities

As already explained in a previous section, the new worker operators are not supplied with capabilities: they can only produce output (or setup notifications) after receiving some input (which comes with its own capabilities).

While we must not issue capabilities for timestamps that have been closed already, it is safe to issue a capability for the *current* timestamp `t`. Here, *current* means the local view of the bootstrap server of the frontier for each operator. Since this local view is a *delayed* view of the global frontier, it is safe to issue a capability for such timestamp. Importantly, the bootstrap server should emit a progress update `(t, +1)` but not the corresponding `(t, -1)`, which should be issued by the new worker instead.

As of now, the new worker cannot produce output out of thin air, which means that their `source` operator are useless (and are actually swapped with `empty` stream under the hood).

6.3 Formal verification of the protocol

There was some interest in formally verifying timely progress tracking protocol, but nobody ever put the effort in doing so. The extra complexity (hopefully not too much) introduced by the bootstrap protocol might make it worth it to have some additional guarantees given by a formal verification. This is not a trivial work and might make an interesting idea for a Master thesis. The protocol (without the rescaling feature) has been described in a document[12].

[12] `https://gitlab.inf.ethz.ch/PROJECT-STRYMON/wiki-progress-tracking/blob/master/report.pdf`