WildFire protocol implementation

Fabio Massimo Ercoli

April 2024

1 Introduction

In this report we will describe a possible synthetic implementation for the WireFire protocol described in the paper *the price of validity in dynamic networks*[1]. For an exhaustive description of the problem setting and all the definitions, please refer to the paper itself or to the slides we prepared to present the project[2]. The idea is to find out an algorithm to reach some consistency guarantee, named *single-site validity*, performing aggregate queries (min, max, sum, count and avg), among a very large and dynamic (high-churn rate) unstructured overlay network, in a crash-stop distributed model, in which we known upper bounds on process execution speeds, message transmission delays, and clock drift rates. While some stronger forms of consistency such as *snapshot validity* or *interval validity* are simply not possible to be achieved in this problem setting, the *single-site validity* can be obtained for instance simply running a very simple but expensive dissemination algorithm called *AllReport*.

1.1 Single-site validity

We can define the *single-site validity* in the following way. A path p_h is called **stable** if all its hosts (h_0, ..., h_k) and all the connections (h_i, h_{i+1}) between them are alive during the time interval [0, T], where [0, T] is the time interval in which the query is executed.
Let H_c be the set of all the hosts h having at least one stable path from h_q (that is the host which invokes the query) and h.
Let H_u be the set of all the hosts h that are alive at some instant in [0, T]. The single site validity is satisfied if:

$$\exists H. \quad H_c \subseteq H \subseteq H_u \quad \text{and} \quad v = q(V) \tag{1}$$

Where v is the result of the query applying the query function q to V, and V is the set of all the values locally-computed by all the hosts in H.

[1] Mayank Bawa, Aristides Gionis, Hector Garcia-Molina, Rajeev Motwani: The price of validity in dynamic networks. J. Comput. Syst. Sci. 73(3): 245-264 (2007) https://doi.org/10.1016/j.jcss.2006.10.007

[2] https://docs.google.com/presentation/d/e2PACX-1vROzkpRvQPwjxz8qMWD3Abibt9yOG_hLyldeUS2yxONSrcCGexbt1IEWM8eP9E_dKk0FqHBx26rJk4/pub

1.2 WildFire protocol

The paper introduces a more efficient algorithm to reach *single site validity* named *WildFire* in which we have two kinds of messages: **broadcast** and **convergecast**.
The algorithm to implement *min* and *max* is the following: The query initiator, named the host H_q, starts the algorithm sending the broadcast message to all its neighbors.
When a host delivers the broadcast message becomes active and it retransmits the broadcast message to all its neighbors.
An active host exchanges and combines its partial query result with its neighbors through the *convergecast* messages. A *convergecast* message is produced and send by some host:

- When a host becomes active, since it can potentially introduce a new value on the system.
- Whenever a host updates its local value, after a delivery of a convergecast message containing an updated value.
- Whenever a stale value is delivered from a neighbor.

Thanks to the time assumptions, we can introduce and use the universal maximum delay value δ to determine when we should terminate the algorithm.
In particular we can demonstrate that as soon as we reach the time $2D\delta$, where D is the stable diameter possibly to be overestimated by a reasonably small constant, we can safely return the distributed query result to the layer above satisfying *single site validity*.

1.3 Continuous queries and approximate single-site validity

The paper also describes how to extend the protocol to cover continuous queries and the remaining aggregation function (count, sum and avg). The problem with those other aggregate functions is that they are **duplicate sensitive**.
The algorithm of the WildFire protocol to implement them satisfies a weaker form of single-site validity, that is only probabilistic, named *approximate single-site validity*.
The details are in the paper and on the slide, we won't cover those cases in this report, since our WildFire implementation will cover only the WildFire for *min* and *max* only, which it has been described in the previous chapter.

1.4 The price of validity

There are algorithms to compute distributed aggregation queries that are intrinsically more efficient in terms of number of messages required to be sent or by number of messages expected to be processed by a random node, the paper mentions the *SpanningTree* and the *DirectedAcyclicGraph*.

On the other hand, they do not satisfy the single-site validity. In particular in case of failures we cannot have any guarantees about the consistency of the returned result.
The price of validity is the performance penalty of the *WildFire* compared with the performance of the *SpanningTree* and the *DirectedAcyclicGraph*.

2 WildFire implementation

In this section we describe our implementation of the *WildFire* algorithm to compute *min* and *max* distributed aggregation queries in terms of final algorithm, technologies used, implementation environment and simulation solution.

2.1 Optimized pseudocode

There are some adjustments that can be applied to the algorithm that seem to be suggested by the same example execution of the *WildFire* protocol presented by the paper.
For instance we can notice that sending a convergecast message to all the neighbors **each time** a node delivers a convergecast message that updates the local value A could be quite expensive.
In our implementation we're going to apply what we can call *a batching strategy*, in which we buffer all the delivered convergecast messages and we process them at regular intervals (δ).
This should limit the overall number of convergecast messages delivered by the system.
Another optimization, suggested by the same paper, is to merge the messages types *broadcast* and *convergecast*, piggybacking the local partial query result value on the *broadcast* message.
In the following you can find the pseudocode of the resulting algorithm. This pseudocode has been the driver of real code used to implement the solution.
As you can see the messages are processed at regular intervals lasting δ and all the complexity is moved to the implementation of the *process-pendings* procedure that is responsible to determine the convergecast messages that should be sent **as a consequence** of the current chunk of the delivered convergecast messages.

upon event $< wildfire,\ Init >$ **do**
    active := FALSE;
    t0 := $\perp$;
    pendings := $\emptyset$;

upon event $< wildfire,\ Query \mid q >$ **do** ▷ Only on node Hq
    active := TRUE;
    t0 := now();
    start-timer(δ)
    D := estimate-diameter()

```
    A := local-compute-query(q)
    for all p ∈ Π do
        trigger < pl, Send | p, [DATA, q, t0, D, A] >
    end for

upon event < pl, Deliver | from, [DATA, q, t0', D', A'] > do
    if now() − t0' < 2D'δ then
        pendings := pendings ∪ {from, q, t0', D', A'}
        if active = FALSE ∧ timer-is-not-started() then
            start-timer(δ)
        end if
    end if

upon event < Timeout > do
    if now() − t0' < 2D'δ then
        process-pendings()
        start-timer(δ)
    else
        active := FALSE;
        trigger < wildfire, QueryReturn |A >
    end if

procedure PROCESS-PENDINGS()                    ▷ Invoked at each turn
    if pendings ≠ ∅ then
        if active = FALSE then
            t0 := pendings[0].t0
            D := pendings[0].D
            A := local-compute-query(q)
        end if
        processes := Π
        for all c ∈ pendings do
            Anew := combine(c.q, c.A, A)            ▷ Combine the results
            processes := processes - {c.from}
        end for
        for all c ∈ pendings do
            if c.A ≠ Anew then
                trigger < pl, Send | p, [DATA, c.q, c.t0, c.D, Anew] >
            end if
        end for
        if Anew ≠ A ∨ active = FALSE then
            A := Anew
            for all p ∈ processes do
                trigger < pl, Send | p, [DATA, c.q, t0, D, A] >
            end for
            active := TRUE
        end if
```

```
        pendings := ∅
    end if
end procedure
```

2.2 Environment and technology

Instead of reinventing the wheel and recreate anything from scratch, we decided to integrate[3] our implementation of the *WildFire* protocol into an open source project named *distributed-framework*[4] that already has several classic distributed algorithms already implemented (consensus, leader-election, ring-topology, ...).
In particular the *distributed-framework* project provides a series of nice utilities to set up, test and benchmark synthetic distributed algorithms:

- Graph builders: to generate different kinds of graphs (such as empty, random, tree, ring, complete, torus, hypercube, hamiltonian, directed, undirected, ...). A graph as usual can be described as a set of nodes and edges, called channels in the context of the framework.
- Generators: to generate unique ids among the entire system (that is used for instance by the vanilla Paxos algorithm)
- Runners and Synchronizers: items that allow us to control the time, in our case we will use them to process the pending convergecast messages at regular intervals.
- Failure generators: tools to simulate crash failure on nodes of the system and evaluate the dependability on a given execution of the algorithm.
- Metrics collectors: number of messages, number of rounds, time passed are stored and provided at the end of the execution to evaluate the performance on a given execution of the algorithm.

The networks generated using the framework are synthetic, each node is not an independent process on an independent machine, as it would be on a real distributed system, but a goroutine, whose real degree of parallelism is determined by the Go-engine. (remember that concurrent $\neq$ parallel).
But thanks to synthetic utilities we mentioned earlier a real distributed-system seems to be pretty well simulated.
On the other hand, it would be very expensive to run and test our algorithm on a real distributed system. Let's imagine how it would cost to run our algorithm for instance on 10,000 machines of any cloud provider.
The framework is written in Golang and this language offers some nice and very model concurrent api. A goroutine is a lightweight thread managed by the Go runtime and it is used to simulate a concurrent node.

[3] See the GitHub pull request: https://github.com/krzysztof-turowski/distributed-framework/pull/49
[4] https://github.com/krzysztof-turowski/distributed-framework

Graph channels are implemented as Go channels and are used to send messages between and synchronize goroutines.
Messages are delivered to nodes using the Go select case statement, which allows a goroutine to wait on multiple communication operations.
In the project Go mutexes are never used, since memory is never shared, but synthetic nodes can only use channels to communicate (as in real nodes).

2.3 Execution simulation

Our program has different parameters that allows the user to customize the experiment.

- n: number of nodes of our generated random graph
- *edgeProbability*: probability of an edge will be generated between any pair of nodes. This value will influence connectivity.
- *faultyProbability*: probability that a given edge may become faulty and if an edge is faulty the probability that may crash at each round
- *query*: the query function to execute, possible values are *min* or *max*
- *diffValues*: local query values will be in the range [0, diffValues-1].
  In particular in case of the *max* query function the greater values will be assigned to the nodes that are farther from H_q according the function:

$$\text{query-local-value(i)} = \lfloor \frac{i \cdot \text{diffValues}}{n} \rfloor \tag{2}$$

  Dually, for the *min* query function, the function will be:

$$\text{query-local-value(i)} = \lfloor \frac{(n-i) \cdot \text{diffValues}}{n} \rfloor \tag{3}$$

  Where i is the node index, i $\in$ [0, n-1] and H_q has index 0.
- *diameterExtimation*: an overestimation of the diameter of the random graph

The output will present the *query result* and the *number of messages* globally exchanged in the system during the time of the algorithm execution.
To run the simulation. Clone the project:

```
git clone -b wildfire git@github.com:fax4ever/distributed-framework.git
```

And execute command similar to following from the project directory:

```
go run example/wild-fire-aggregation-query.go 5000 0.01 0.2 max 50 2
```

In this case we execute the algorithm on a graph of 5,000 nodes, with an emphedgeProbability of 0.01, with a *faultyProbability* of 0.2, the *query* is a *max*, the *diffValues* is 50 and the *diameterExtimation* is 2.

For instance this is the result prompted from my machine on a specific execution (results may vary since the graph is a random graph):

```
2024/04/16 10:42:43 Round 4 finished
2024/04/16 10:42:44 Total messages:  224003
2024/04/16 10:42:44 Total rounds:  5
Query result: 49 . #(messages):  224003
```

Alternatively, you can run the automated test:

```
go test -timeout 30s -run ^TestWildFireAggregationQuery$ \
github.com/krzysztof-turowski/distributed-framework/test
```

In this case the parameters can be changed only by changing the test class. Finally, a different method to execute the simulation is to compile the project:

```
go build example/wild-fire-aggregation-query.go
```

and run the compiled file as many times you want, optionally with different parameters:

```
./wild-fire-aggregation-query 50000 0.001 0.2 min 100 4
```

The following is a possible outcome:

```
2024/04/16 10:45:50 Round 8 finished
2024/04/16 10:45:51 Total messages:  2491160
2024/04/16 10:45:51 Total rounds:  9
Query result: 0 . #(messages):  2491160
```

3 Comment on the achieved results

3.1 Dependability evaluation

In this section, we report the outcome of some experiments similar to the ones presented in the paper chapter *Achieving Single-Site Validity*, in which we also try to evaluate the quality of the result evaluated on different **degrees of dynamism** in the network.
For these settings we will try a *max* function on a synthetic graph of 50,000 nodes, in which each node has a local-computed-query value that is exactly equal to the node id, so it will be an integer number from 0 to 49,999. The *edgeProbability* for this test is 0.00004, this means that the expected value for the number of edges of a random node is 2.
We retry the execution under different degrees of dynamism changing the *faultyProbability* and we report the result on the following table. The results have no statistics validity, since we report only a single run for each parameter set.

If you want to run the same test on your machine type the following command:

```
./wild-fire-aggregation-query 50000 0.00004 0.2 f 50000 3
```

replacing f with different *faultyProbability*s. For instance: 0.1, 0.2, 0.4, 0.9, 0.99, 0.999.
These are the results we get in our execution.

faultyProbability(f)	**Query result**	**#(messages)**
0.2	38639	9
0.4	47236	159
0.8	48149	169
0.9	40789	597
0.99	49046	31430
0.999	0	0

We can see that usually despite the failures the query results are usually very close to the optimal value of 49,999. This result may suggest the validity of the algorithm.

3.2 Performance evaluation

We're going now to run some tests to have an idea about the order of the messages that are exchanged between the nodes during the execution of the algorithm.
In this setting we kept constant the *faultyProbability* to 0.2, while we change the size of the graph *n*. We also change the *edgeProbability* to try to keep constant the expected node connectivity to 4. We test both the min and the max. The *diffValues* is kept fixed at 100 and *diameterExtimation* is kept fixed at 4 for simplicity, that is supposed to be a great enough value. The command would like the following:

```
./wild-fire-aggregation-query n m 0.2 q 100 4
```

#hosts(n)	**edgeProbability(m)**	**query(q)**	**Query result**	**#(messages)**
1024	0.003906	max	99	5182
1024	0.003906	min	0	3241
2048	0.001953	max	99	8991
2048	0.001953	min	0	8943
4096	0.000977	max	99	14909
4096	0.000977	min	0	13751
8192	0.000488	max	99	33207
8192	0.000488	min	0	32553
16384	0.000244	max	99	57262
16384	0.000244	min	0	48489

The result would suggest the idea that for this specific setting the number of messages could be proportional to the number of the hosts, in this case by a factor of 4, that would coincide with the number of the edges. The performance seems to confirm the linearity of the complexity of the algorithm with respect to the size of the input.
But, again, the results reported have no statistics validity, since we report only a single run for each parameter set.