This is a follow up of Sensor Fusion for BFT Clock Sync, Time as a Public Service in Byzantine context and

Time attacks and security models writings.

The goal of the write up is to design protocol architecture to address clock sync problems in BFT context. Clock sync protocols can be rather simple, but it can be hard to reach precision and accuracy guarantees when arbitrary faults are possible and there are lots of participating nodes. There are three factors, which can help to solve the problem in practice: a) reference clocks, synchronized to a common Time Standard, b) independent local clocks and c) uncertainty of clock drift and network delays.

Protocol requirements

There are two group of requirements to the protocol:

- precision and accuracy bounds, i.e. time estimates of participants should be close to each other (precision) and close to the common time standard (accuracy)
- scalability, we assume there can be thousands and tens of thousands of participating node

The requirements are discussed in more detailshere.

Design requirements

However, reasoning about time in distributed systems can be difficult, e.g. establishing worst case bounds can be non-trivial. It can become even more complicated, if new features added to the protocol to solve particular problems (e.g. found during simulations).

We therefore formulate two additional requirements on the design level

- protocol should have clean and extensible structure
- it should be relatively easy to reason about the protocol properties (precision, accuracy, scalability, resources)

“Mathematical” design

Reasoning is one of the most important problems. In general, we are usingSensor Fusion Framework, which establish connections to already known methods and their properties. Particularly, we employ the following “tools” and simplifications:

- intervals and interval arithmetic, to model uncertainty
- three sources of uncertainty and faults: a reference clock, a local clock and network delays
- single kind of fault: a true value is beyond interval bounds (of estimate of the value)
- interval fusion procedures (robust interval aggregation)
- theorems about interval fusion and its properties

Uncertainty, faults and intervals

Even correctly functioning devices are susceptible to fluctuations: clocks can output readings meandering about true value and network can delay messages for varying periods of time. This directly affects time-related operations.

The uncertainty is inevitable, but can be bounded. We represent the uncertainty with intervals. And assume a fault to occur only when an output value is beyond admissible bounds. As a consequence it allows to hide certain faults implicitly, when summing up several intervals, i.e. a reference clock can go wrong, but network delay can compensate this in some cases, so the resulting estimate can still include true value.

We assume three sources of uncertainty - and faults, correspondingly:

- reference clocks, which output intervals that should containing true time (as defined by a common time standard).
- local clocks can be drifting, but drift rate is limited, i.e. clock skew belongs to an interval around 1 (second per second).
- network delays messages, but the delays in normal conditions should belong to an interval too.

All other sources of uncertainty can be bounded by corresponding intervals, however, we ignore them (e.g. they can be subsumed by a network delay interval).

As we assume that there is only one kind of fault - a value doesn't belong to an interval (that it should belong), other faults should be mapped to the fault kind (see here for more details). We are assuming a limited amount of faults at the same time, e.g. < 1/3 typically. Intervals can be widened to reduce probability of a fault, but this will lower accuracy and/or precision.

Mathematical goals

There are two main properties that we care about:

- an interval contains true value
- an interval width, i.e. the estimate uncertainty meets the required bounds

As we are assuming reference clocks, synchronized to a common standard, the most important value kind is the time as understood by the common standard (e.g. UTC).

The second property ensures that time estimates are useful, otherwise one could choose a very wide, but uninformative interval.

The properties are assumed to be hold for input data and sensor fusion procedures should preserve them, so that they hold for outputs too.

Reasoning about intervals

In BFT context, however, faults are possible, so we assume the properties hold for majority of cases, but not for all of them. Sensor fusion procedures therefore should be robust and preserve the properties for output result despite partially incorrect inputs, under the appropriate majority assumptions.

We need an instrument to reason about such properties. With intervals and the single fault kind, the reasoning actually becomes rather simple: if majority of estimates are correct, then we can construct an interval which is correct too (see here for more details), i.e. it contains the true value and the uncertainty is bounded. This is the main "theorem", which is to be discussed more rigorously and formally in a separate write up.

Weaker forms of the theorem are possible, which will gives reasonable estimates, even if such an interval doesn't exist. For example, one can state that a lower boundary of interval is wrong with less that 1/3 cases and the same holds for upper bounds. Thus, we can also construct an interval, containing the true value. In other words, even "honest" clocks can go faster or slower, but only a third minority are wrong the same way. So, there is <1/3 slow clocks, < 1/3 fast clocks and > 1/3 true clocks.

An additional important property, which is important if an iterative/recursive estimation is desired, is that the estimate fusion process is a contraction, i.e. it shrinks estimates. Then, it can be applied iteratively, e.g. to get better estimates or to employ implied timestamps. Otherwise, output uncertainty can grow and becomes useless at some point.

Estimate model

In the presence of reference clocks, thus, the main goal for a honest node is to obtain most accurate true time estimate and exchange them in most accurate way possible.

Any correct estimate is modeled by $E(t) = t + acc(t) + net(t) + drift(t)$

. Where $acc(t)$

, $net(t)$

and $drift(t)$

contain zero. Therefore, a correct estimate contains t

.

A fault is modeled by adding an additional unbounded term $e(t)$

, which is zero (or small) for correct estimates and arbitrary for faulty ones.

Framework

If majority of estimates are correct, then we can construct a correct estimate. But it should be tight enough to be useful. In general, precision and accuracy bounds are proportional to the sum of reference clock, network delay and local clock drift

uncertainties. If faults are uncorrelated, then bounds should be tight, however, an attacker may construct correlated faults, so that worst case bounds are reached.

Measures can be taken to improve the bounds, e.g. better estimates are known and.or can be constructed. However, the additional information can be costly to communicate when there are lot of nodes.

In the section, we explore which data is available and required for constructing time estimates. And which processes are needed to produce final time estimates.

Data and time sources

Raw data is obtained from time sources, either reference clocks

(RC) or local clocks

(RC). We assume all nodes possess at least one reference clock and at least one local clock. These constitute local time sources

. A node can fuse its local time sources and produce a local estimate

.

NB

In general, some nodes can operate without any reference clock, since they can rely on the estimates received from peers. That can be modeled as if such a node possesses a faulty RC.

Besides own local data, nodes can send their data to peers and receive it from them. The data received from peers constitute remote time sources

. Remote and local time sources are then combined to produce final estimate

. In general, it can be done in an iterative/recursive manner, i.e. final estimates obtained at round N can be used as input data in consequent rounds.

Historical data can be used to improve estimates, e.g. to calibrate clocks and/or filter out erroneous readings.

Besides time data, there are parameters, which are very important to achieve good accuracy and precision, e.g. reference clock accuracy, local clock drift and network delay bounds.

Reference Clocks

Correct reference clocks are synchronized to a common time standard. Let’s denote the un-observable “true” time, predicted by standard, with small t

letter. It’s predicted with reference clocks, up to some offset and scale (prescribed by the common time standard), which do not matter.

We model a Reference clock

as a function from real time to an interval, i.e. R_q(t)

mean an estimate of a reference clock q

at the point of real time t

. If the clock is correct, than t \in R_q(t)

.

local clocks

A node have an immediate access to an interval local clock, i.e. they have a un-observable offset to a reference clock. The clock rate also can drift, however, for a correct local clock, the drift is bounded.

We model local clocks in a similar way, i.e. it’s a function from real time to a point estimate (in general, it’s an interval too).

The property of a correct local clock q

is that it can estimate intervals of time, e.g. \exists \rho \gt 0 \forall t_1, t_2 {1 \over (1+\rho)}(t1-t2) \le |L_q(t_1)-L_q(t_2)| \le (1+\rho)(t_1-t_2)

. An immediate consequence, is that real time interval is bounded by local measurements in the same way, i.e. {1 \over (1+\rho)}(L_q(t_1)-L_q(t_2)) \le |t_1-t_2| \le (1+\rho)(L_q(t_1)-L_q(t_2))

.

Using interval arithmetic, L_q(t1)-L_q(t2) \in (t1-t2) * \delta

, where \delta

is interval [{1 \over (1+\rho)},(1+\rho)]

.

Time operations

In order to construct reliable estimates in presence of faults, nodes need to exchange their clock readings and/or estimates - to perform time transfer

. However, they cannot do that too often, especially assuming, there are lots of nodes. So, nodes need to perform time keeping

in between, i.e. to update their estimates, as time passes. In BFT context, remote clock readings can be faulty including Byzantine faults. So, a node should filter

out erroneous readings. In general, we consider filtering

as a part of sensor fusion

, since robust sensor fusion

procedures explicitly filters out outliers.

Nodes' clocks may have varying properties (clock accuracy, stability, drift, etc). In order to reduce amount of data to exchange, a common standard is assumed, which is necessarily conservative. Real devices can be more accurate. Thus, devices can be calibrated

and tighter bounds can be used, leading to improved accuracy and precision of final estimates.

Time keeping

Let's assume a node p

obtained at time t_1

a time estimate T(t_1)

(e.g. read a reference clock or received over a network from another node). Later, at time t_2

it can calculate an updated estimate, using its local clock. Assume also, that it also measured its local clock L(t_1)

at t_1

. Then, later, at time t_2

, it can estimate T(t_2) = T(t_1) + (1+\delta)(L(t_2)-L(t_1))

, where the 1+\delta

term accounts for the local clock drift during the period. In general, it's more correct to call it a q

's estimate of T

at time t_1

, i.e. E_q T(t_2)

, where E_q

denotes 'estimate by q

'. For example, process q

's estimate of process p

time can be expressed as E_q E_p T(t)

.

Time transfer

Nodes can access remote clocks via a message exchange, which entails a message delay. The delay is unknown, but for a correctly functioning network at the moment of message transmission, we assume the delay is bounded, which we denote with \Delta

letters. It can be path dependent, i.e. we use \Delta_{pq}

to denote a delay bounds when sending a message from p

to q

.

We model time transfer in the following way:

- process q

maintains an estimate of real time, that can be a reference clock reading or a derived estimate. Typically, we assume that it's a real time estimate, i.e. t \in E_q(t)

, if the process is correct around t

.

- at the moment t_1

, q

sends its estimate E_q(t_1)

to process p

.

- p

receives the message at time t_2

and records local time L_p(t_2)

. At t_2

, p

can estimate E_q(t_2) = E_q(t_1) + \Delta_{qp}

, as we assume that network delay t_2-t_1 \in \Delta_{qp}

. We assume that clock drift over the period is negligible (it can be incorporated into \Delta_{qp}

).

So, process p

can use the above time keeping

formula, to obtain an estimate at any time t

, process p

: E_q(t)

by calculating an interval E_q(t) = E_q(t_1) + \Delta_{qp} + \delta*(L_p(t)-L_p(t_2))

.

Filtering

Erroneous reading can be filtered out. For example, a reference clock is accurate with \pm 500ms

, then two clock readings can be 2 seconds apart. Additionally, local clock can drift, for example, two ref clock readings are performed with 10000 seconds interval. Local clock can drift \pm 10000 * 0.0001 = \pm 1 sec

. So, if two such ref clock readings are more than 3 seconds apart, then one of the is wrong. If there is a history of reference and other clocks readings, then a minority of erroneous readings can be filtered out.

In general, we consider filtering to be a part of robust sensor fusion. Though, it may be beneficial to consider filtering explicitly in some cases.

Sensor fusion

There exist numerous procedures for robust sensor fusion (see exampleshere). We mainly use Marzullo's algorithm, which is simple and worked best in our experiments, though simulations and analysis with other fusion procedures are to be done too.

Calibration

Calibration of local clocks can improve accuracy of estimates dramatically. Network delays between nodes can be measured and calibrated too.

As resulting uncertainty depends on input data uncertainties and nodes can keep historical data, it can be highly desired to build more accurate models of clocks and network delays.

The general idea is that Theil-Sen or repeated median regression can be adapted to the interval settings and thus can be used to robustly estimate simple linear models using interval historical data.

So, a node can produce local estimates

which filter outs outlier reference clock readings, with respect to historical data. The same approach can be applied to filter out outlier peer time estimates. In result, a node produces its final estimates

using not only local and remote time sources, but historical records too. Which gives greater abilities to tolerate periods when amount of faults exceeds expected level, for a reasonable period of time (say, less 1/3 or 1/2 of the period that historical data is kept in memory)

The approach can be difficult to implement and to assess. We therefore leave discussion of details until further writings.

Design choices

In theory, there can be multiple ways, how clock synchronization performance and fault-tolerance can be improved in theory. In practice, however, there are bandwidth limitations. Which are critical in our case, since we are assuming tens of thousands nodes,. In the section, we overview main design choices and their consequences.

Protocol kind

We know three main ways, how BFT clock sync protocol can be implemented:

- adapted BFT agreement
- iterative In-exact/Approximate Agreement
- non-iterative In-exact/Approximate Agreement

Adapted BFT

There are clock sync protocols which are adapted versions of BFT Agreement protocols. We do not consider the option, since they require lot of communication and because of circular dependence problem: our clock sync protocol is intended to harden BFT protocols, which depend on clock synchronicity assumption.

However, reliable exchange of parameters may require a BFT protocol.

Iterative

There are Approximate/In-exact BFT Agreement protocols (see detailshere), which are iterative: i.e. participants have initial

estimates and they exchange with them iteratively, reducing discrepancy between honest node estimates.

As we assume reference clocks, synchronized to a common Time Standard, iterativity is not required. However, it can be a helpful to either improve accuracy and precision of estimates or to reduce network traffic.

If the clock sync protocol is a complement to some other BFT protocol, operating in a lock-step fashion (like Ethereum 2.0 beacon chain protocol), then the existing message flow can be employed to transfer clock sync data. A particularly promising case is to use implied timestamps of the message flow, if such protocol prescribes that certain messages should be sent in a pre-defined moments of time. However, such moments of time will be defined relating to final estimates

in most cases. That means the time estimation is iterative/recursive.

Non-iterative

Non-interative fusion of clocks is the most simple option, though it requires to transfer local estimates

(and/or other local data). In particular, it does not require fusion procedure to be a contraction, so it can give reliable estimates in a broader set of situations.

Therefore, we choose it as the main option, which can be augmented with iteration and/or BFT protocol for parameters exchange.

Data transfers

Parameters

In order to calculate final estimates, nodes need to know bounds of reference and local clock accuracies and network delays. The data can be specific to a particular clock or network path and/or can vary. Exploiting tighter bounds can be beneficial for accuracy and precision. However, it can be very costly to transfer the amount of data.

The main option is therefore assume predefined estimates, which are necessarily conservative. Or refresh them relatively rare.

In some cases, tighter bounds can be obtained by using calibration

using historical data.

Relaying peer's data

Relaying other nodes data can be very helpful to fight Byzantine Faults, for example, a faulty node can send different estimates to different peers (that can be a consequence of varying network conditions too). If nodes relay data, received from peers, that can help to filter out certain faults.

However, it can easily overload network resources.

In a p2p network, nodes transit messages, so it's useful to employ data relaying in a limited form. For example, a node p

can receive clock readings from node q

via different routes, due to message routing/diffusion in p2p-graph. So, this is the additional information, which is available "for free" for the clock sync protocol, given that it's already necessary in some other protocol.

Local data

There is also a number of variants, which local data can be sent to peers.

All available data

A node can have several local sensors, i.e. one or several local clocks and one or several reference clocks.

NB

We assume reference clocks to be local resources, i.e. specific to a particular node, though in practice, it's an interface to receive data from a remote clock, e.g. GNSS satellite or NTP server. Transferring the info can be beneficial for accuracy and precision, but will require lot of traffic.

Local estimates only

To reduce traffic, instead of sending all its info to other nodes, a node can send its local estimate only, which is a fusion of

local and reference clocks.

Final estimates

As pointed out earlier, a node can send its final estimates

only, which is a fusion of local data sources together with data, received from other nodes. That can reduce network traffic, if the node participates in another protocol, where is should send messages at predefined moments of time.

Thus, recipients of such messages can calculate implied timestamps of when the message had to be sent. As a result, protocol can piggyback existing message flow with zero-cost overhead, which can be very important in some cases.

For example, in Ethereum2.0 beacon chain protocol, attestation messages are to be aggregated. Aggregation format is very efficient, i.e. one bit per attester is used to denote attestation producer and attester signatures are aggregated using BLS signature aggregation scheme. Adding just one new field to the attestation messages will increase the size of an aggregate considerably.

Implementation

Before discussing particular implementation strategies, let's overview general assumptions and terminology.

Node-level assumptions

Each node:

- has one or several reference clocks (RC)
- one (or several) local clock (LC)
- maintains a local estimate (LE), by fusing its local data (RC and LC)
- maintains a final estimate (FE), by fusion of data received from others node and local data (e.g. LE, or RC+LC)
- exchanges estimates with peer nodes (PE)

Network-level assumptions

Connectivity and membership:

- membership is static (will extend to dynamic membership in future)
- communication graph is p2p, though it can be fully connected in a simple case

Parameter-level assumptions

Parameters:

- three groups of interval bounds:
- network delay, can be the same (for all paths) or path dependent (e.g. \Delta_{pq}

)

- reference clock accuracy, can the same (for all nodes) or node/clock dependent
- known to nodes, need not be estimated typically
- known to nodes, need not be estimated typically
- local clock drift, the same for all nodes typically (e.g. 100 ppm

) * real (vs nominal) clock drift can be better (less magnitude), it can vary from node to node and can be estimated. We assume it's not known to nodes, though can be specified during simulations

- real (vs nominal) clock drift can be better (less magnitude), it can vary from node to node and can be estimated. We assume it's not known to nodes, though can be specified during simulations
- network delay, can be the same (for all paths) or path dependent (e.g. \Delta_{pq}

)

- reference clock accuracy, can the same (for all nodes) or node/clock dependent
- known to nodes, need not be estimated typically
- known to nodes, need not be estimated typically
- local clock drift, the same for all nodes typically (e.g. 100 ppm

) * real (vs nominal) clock drift can be better (less magnitude), it can vary from node to node and can be estimated. We assume it’s not known to nodes, though can be specified during simulations

- real (vs nominal) clock drift can be better (less magnitude), it can vary from node to node and can be estimated. We assume it’s not known to nodes, though can be specified during simulations
- adversary power limitations
- unforgeable signatures
- can control a minority of nodes (or their clocks), typically, \lfloor {(n-1) \over 3} \rfloor

or \lfloor {(n-1) \over 2} \rfloor

- unforgeable signatures
- can control a minority of nodes (or their clocks), typically, \lfloor {(n-1) \over 3} \rfloor

or \lfloor {(n-1) \over 2} \rfloor

Initial design

We start evaluation of the approach with the most simple and lightweight approach:

- one RC and one LC per node
- LE is maintained by simple time keeping (no filtering, calibration, etc)
- exchanged estimates are either LE or FE (last one results in a form of recursion)
- FE is a fusion of LE plus remote estimates (peers’ LE or FE)
- n-to-n connectivity, nodes communicate by direct channels
- membership is static
- all parameters are known beforehand

Viability

Below is a sketch proof that the approach is viable. A more formal and rigorous proof with worst case bounds analysis is to be described in a future work.

A correct node can maintain a LE estimate (i.e. latest RC reading adjusted with elapsed time): LE(t) = RC(t_1) + \delta * (LC(t) - LC(t_1)), s.t. t \in LE(t)

, if no fault occurred.

A correct node (p

) can maintain estimates of LE’s (or FE’s) of other nodes (q

): E_p(LE_q(t)) = LE_q(t_1) + \delta (L_p(t) - L_p(t_1)) + \Delta_{qp}, s.t. t \in E_p(LE_q(t))

, if no fault occurred.

Under the third minority assumption (total amount of faulty intervals \le \lfloor {(n-1) \over 3} \rfloor

), a correct node can calculate a fused interval, which still contains true time.

The width of the interval is less than max width of correct intervals, if the fusion procedure is a contraction. That can be important if nodes exchange with FE kind of estimates. However, if the width expansion is limited, it can be okay, if nodes exchanges with LE kind of estimates. The last can be beneficial, since fusion procedure can give meaningful results in a broader set of cases (e.g. when almost half of reference clocks are faulty).

NB

Care should be taken in the case of recursive estimation sent over network, since network delays widen the estimate interval. To the best of our knowledge, that can be handled by some sensor fusion approaches and our preliminary experiments support this. We will investigate this issue in more details and more rigorously in future works.

Implementation code

We have implemented the simple protocol as well as simulation code in Kotlin aiming to obtain simple and concise code, which is easy to analyze. The code is available in here.

Preliminary simulation results

We have simulated a simple attack, when an adversary can control some fraction of node's reference clocks (emulates NTP attacks).

Using Marzullo's algorithm, the protocol can tolerate big clock offsets, incurred by adversary for < 1/3 or < 1/2 of faulty clocks, depending on algorithm settings.

Simulation parameters:

- N=1000 amount of nodes
- ref clock accuracy \pm 500 ms
- network delay \Delta = [0, 2000ms]
- (N-1)/3 or (N-1)/2 faulty clocks
- k (expected maximum amount of faults) = (N-1)/3 or (N-1)/2
- one- or two-sided attack (i.e. adversary shifts faulty clocks in the same or in both directions)
- faulty clock offset - 10000ms

EF correct means FE interval contains true value

Precision means discrepancy between nodes' FE midpoints.

Offset means approximate offset of FE (should be less 1500ms, to be correct).

attack type

faults

k

FE is correct

offset

precision

no attack

0

333

yes

\pm

100ms

~170ms

one-sided

333

333

yes

200ms

~200ms

one-sided

499

333

no

~5000ms

~200ms

one-sided

0

499

yes

\pm

100ms

~200ms

one-sided

333

499

yes

~600ms

~240ms

one-sided

499

499

yes

<1500ms

~250ms

two-sided

666

333

yes

\pm

100ms

~300ms

Resulting time estimates remain correct under attack (i.e. resulting interval includes true time), though accuracy becomes worse, but within expected bounds< RC accuracy + max delay

/ 2, with the exception of the case, when amout of faults exceeds k

(i.e. maximum expected amount of faults, which is normal). Discrepancy between nodes' estimates (i.e. a form of precision measure) doesn't change much, even under severe attacks.

An interesting moment, is that protocol can tolerate about 2/3 of faulty clocks, if < 1/3 of clocks are too fast and < 1/3 are two slow. Precision loss is moderate and estimates remain accurate on average (quite similar to no attack case).

Besides attacks, network delays seem to be a major factor affecting precision, since network delays are Byzantine to some degree, i.e. when a node broadcast its estimate to peers, different peers can receive the message with varying delays (including message omissions modeled as being delivered in some distant future).

Simulation details is to be described in an upcoming write up.

Further improvements

The initial design can be improved:

- several reference clocks can be used (typical NTP setup includes 4 NTP servers), as well as several local clocks
- historical data can be employed:
- skew of own local clock can be estimated from data
- skew of peer local clocks can be estimated too
- actual network delays can be estimated
- outlier reference clock readings can be filtered out
- outlier peer time estimates can be filtered out
- skew of own local clock can be estimated from data
- skew of peer local clocks can be estimated too
- actual network delays can be estimated
- outlier reference clock readings can be filtered out
- outlier peer time estimates can be filtered out
- p2p graph support
- clock readings received via different routs can be supported (e.g. employing transient traffic)
- network delay along paths can be estimated (if a message contains information about the path)
- clock readings received via different routs can be supported (e.g. employing transient traffic)
- network delay along paths can be estimated (if a message contains information about the path)

We are going to implement and evaluate the designs in near future.