416 Distributed Systems: Assignment 2 [Traced nim, fcheck lib, Robust nim]

Due: February 15 at 6pm PST

Winter 2022

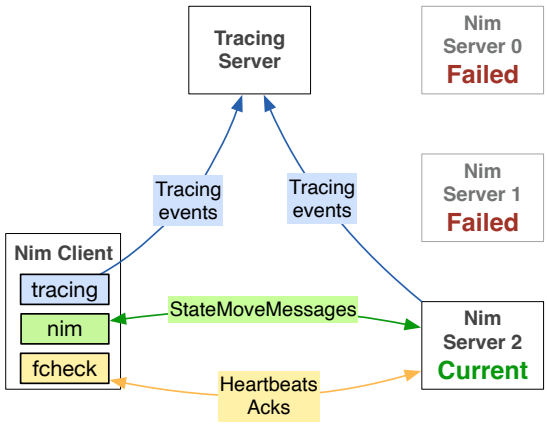
51902c0

This is a multi-part assignment that builds on the the nim client that you have developed in A1. By the end of this assignment, your nim client will be able to cope with server failures and you will be able to visualize an execution of your system across multiple nodes. Note that this is the last solo assignment (A3 and the projects will require team work). In this assignment you will work on three parts:

1. Add *distributed* tracing support to your nim client by using tracing tokens
2. Create a UDP-based failure checking (fcheck) library to detect server failure
3. Extend your nim client to detect server failure with the fcheck library and fail-over to a different nim server

The above three parts require a working nim client. If you want, you can use our version of the nim client for this assignment. But, you can also use your own. As with A1, your code will be marked automatically (we will write servers and script scenarios to exercise your fcheck library and updated nim client). It is therefore important to follow the spec below exactly, including the distributed tracing API, the `fcheck` API and its semantics, the UDP protocol/packet format, and other details.

Assignment overview



Part 1: Distributed tracing support. So far, you have been tracing local actions from your nim client. The resulting trace is a totally ordered trace of client-side actions. It is useful for understanding nim client behavior, but it does not provide visibility into the rest of the system. In this part of A2, you will add distributed tracing support, which will allow the tracing server to observe tracing events not just from your nim client, but also from the nim servers that your client communicates with. You will also be able to visualize the resulting trace using the ShiViz tool. Introducing distributed tracing support will require a change to the packet format and in how you use the tracing library.

Part 2: Failure checking library. Distributed systems are frequently designed to deal with failures. There are many kinds of failures (hardware/software/network/etc). In this assignment you will focus on *node failures*. In this part of A2 you will design a library for failure detection (fcheck). This library will monitor a remote node and will be able to notify the local library client when the remote node has failed. Note that real distributed systems, including those you will build in this course, operate over networks that are asynchronous and unreliable. This makes true node failure detection impossible: you cannot distinguish the failure of a node from the failure of the network. Therefore, your fcheck will provide *best effort* failure detection. You will structure fcheck as a library that can be re-used across projects (you will use it in A3). Your library must integrate a simple round-trip time estimator and use a UDP heartbeat protocol.

Part 3: Dealing with nim server failures. In A1 your client code assumed that the server does not fail. We will remove this assumption in A2. Once you have built the fcheck library, you will use it in your nim client. Specifically, you will (1) use

fcheck to monitor the current nim server that your nim client is connected to, (2) use fcheck to detect when this nim server has failed, and (3) add fail-over behavior to your client to use a new nim server when the current nim server has been detected as failed. After failing over to a new server, your nim client should begin monitoring the new server with fcheck, and fail-over again if necessary.

The three parts above build on one another: the fcheck library must support distributed tracing and record certain actions, and the failover process must use the fcheck library. We recommend starting on a part i+1 only when you have completed part i. The new nim servers that we will allow you to test against will help you implement each of the parts. These nim servers all expect distributed tracing. You can use a non-failing nim server to test your work in part 1. To test your fcheck from part 2 you can use a non-failing nim server and a nim server that sometimes fails and restarts. Finally, to test part 3 you can use a collection of nim servers, each of which eventually fails and restarts.

1.1 Distributed tracing support

We begin with a description of distributed tracing because this will be a powerful new tool in your debugging toolbox. You will also use distributed tracing in A3 and in P2, so it's a good idea to understand it well. You will retain the basic nim protocol, but you will augment it with distributed tracing. What this means in practice is that the UDP messages will include some additional information. You will also need to add more tracing library calls/logic to your code.

Distributed tracing, which is available in the same tracing library you used in A1, introduces two new features: (1) the notion of a *trace* with which recorded actions are associated, and (2) vector clock timestamps to track the happens-before relationship (if any) between actions across nodes. These two features require a slightly different way of thinking about tracing in your system.

You already know that a *trace* is a set of recorded actions that are associated with a unique *trace ID*. With traces, actions must be recorded as part of traces. So, you must first get access to a trace and then you can record an action (i.e., `Trace.RecordAction(action)`). So far, you have been using `tracer.CreateTrace` to get a trace instance, but how do you get two different nodes to record an action to the same trace? The answer is tokens.

To allow another node to record an action into a trace, a trace must be serialized into a *token* (using `trace.GenerateToken()`), which can be sent to another node, at which point the receiving node can reconstruct the trace instance from the token (using `trace = tracer.ReceiveToken()`). You will typically serialize a trace multiple times since *every token must be unique* and a trace created by your nim client will span several nodes. More concretely, to generate the token from the current trace your code would call `Trace.GenerateToken()`, which will return a token. Your code can then include this token in a UDP packet (one reason why the packet format has to change). On receiving a token, the node can reconstruct the trace by calling `Tracer.ReceiveToken(token)` on the received token.

So, in summary, to implement tracing across nodes your code must (1) serialize a trace into a token for another node (nim server) to use, (2) receive an existing trace from another node (nim server) in the form of a token, and (3) deserialize the received token into a trace instance that you can use.

For a detailed description of the distributed tracing API, please see the go doc documentation for the tracing library.

1.2 Message Format

In A2 the [StateMoveMessage] is updated to look as follows:

```
type StateMoveMessage struct{
        GameState []uint8
        MoveRow int8
        MoveCount int8
        TracingServerAddr string
        Token TracingToken
}
```

In each message: [GameState], [MoveRow], and [MoveCount] have mostly identical semantics as in A1. The key semantics change is that during fail-over to a new nim server, the client should send its last valid move with the appropriate GameState, MoveRow, and MoveCount fields values to the new nim server. That is, during fail-over mid-game the client **should not** send an opening message to the new nim server. When starting a new game, however, the client must send an opening move message, as before in A1.

There are two new tracing-related fields in the `StateMoveMessage` struct above:

- `TracingServerAddr`: contains a string representation of the ip:port of the tracing server to connect and trace actions related to this message.
- `Token`: represents a trace ID that the recipient should deserialize (using `tracer.ReceiveToken`) (after connecting to the tracing server) and with which the recipient should then associate tracing actions.

In this assignment the nim server will use the tracing server specified by the client in each StateMoveMessage. It is important that the `TracingServerAddr` and `Token` fields are both set correctly in **every** message to the nim server. If these fields are incorrect -- e.g., if the nim server cannot connect to the specified tracing server or cannot deserialize the token, then the nim server will either (1) respond with its previous move, or (2) not respond if this is the first message that the nim server has received from this client.

Because the nim server uses the tracing server that the client tells it about, `StateMoveMessage` packets **from** the nim server will specify an identical tracing server in the `TracingServerAddr` field. However, the `Token` field will always be different. The client, must deserialize the received token right after receiving a StateMoveMessage from the nim server, and the client must use the resulting trace object (from the deserialized token) in its later tracing commands.

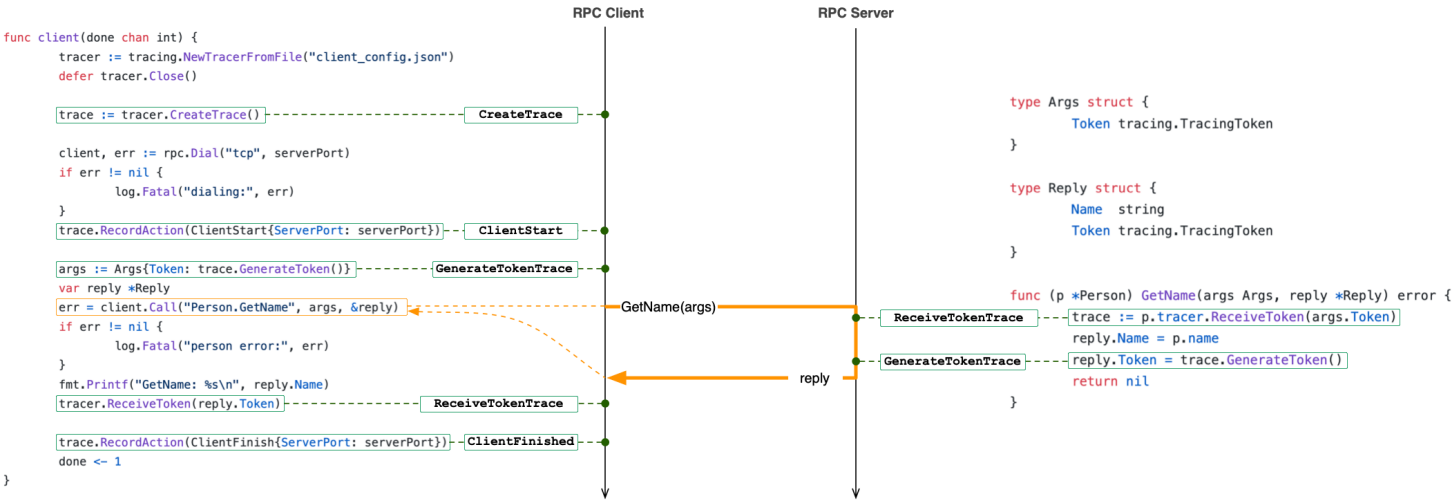
Note that in this assignment you will augment the nim protocol from A1 with distributed tracing. You will not, however, trace the messages sent by the fcheck library. Though you will record several new actions related to failure detection.

1.3 End-to-end example of distributed tracing + ShiViz

A key feature of distributed tracing is that now all actions include not just physical timestamps but also vector clock timestamps. These vector clocks timestamps allow us to reconstruct the happens-before relation between recorded actions. This relation will look like a DAG with vertices associated with different nodes in your system. The tracing library's output supports a visualization tool, called ShiViz, that you can use to visualize the recorded DAG of events.

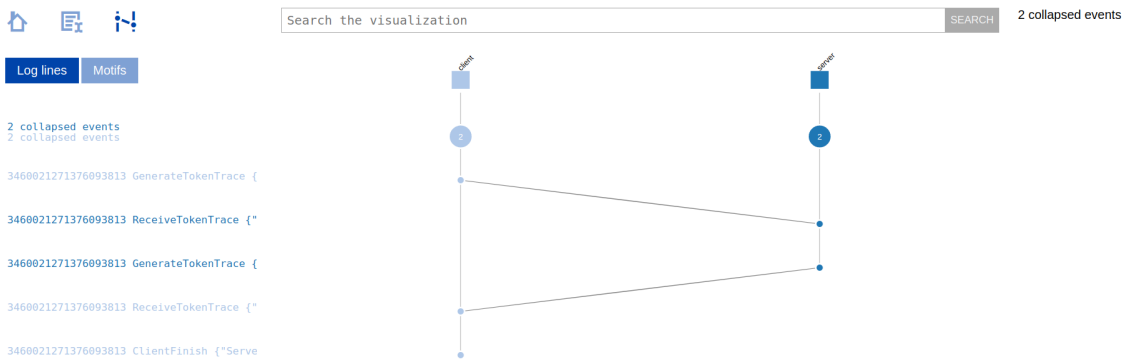
Let's consider a simple example. This example consists of a client and a server. This example uses RPC, but it is identical in spirit to the UDP description above. The client invokes an RPC on the server and then the server returns the response to the client. You can find the implementation here. This implementation also illustrates how to use the distributed tracing API described above. To run this example, you can run `go run main.go` from the example/client-server directory.

Notice the use of distributed tracing in main.go of the above example. The caller generates a token, passes it to the `GetName` RPC function as part of the `Args` struct. The callee runs `GetName` RPC and in the line just before returning generates a return token (**ret-token** in the RPC specs below) and returns this token as part of its return value in the `Reply` struct. The two tokens, one in the forward direction and one in return direction, allows us to reconstruct the happens-before partial ordering of the distributed execution. The diagram below illustrates this in more detail:



The tracing server also generates a ShiViz-compatible log that can be used with ShiViz to visualize the execution of the system. This log file is defined in the tracing server configuration file as `ShivizOutputFile`, which has the `"shiviz_output.log"` value by default.

We can upload the generated ShiViz log, i.e., `shiviz_output.log`, to the ShiViz online tool. For this, click on the blue "Try out shiviz" button and select a file to upload, choosing your local `shiviz_output.log` file. You should then see the following diagram that you can interact with:



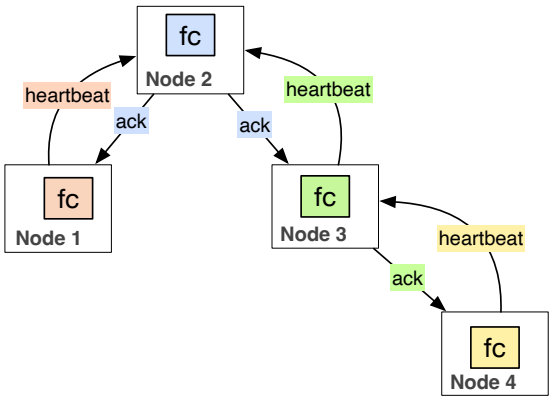
2.1 Detecting failures with the fcheck library

In this assignment, the nim server your client communicates with may fail. However, the A1 client protocol continuously re-transmits a message that has not been replied to every 1 second, indefinitely. The client needs (1) a way to detect that the server has failed, and (2) needs to respond to server failures. The fcheck library that you must develop is used for failure detection. This is the focus of part 2 of A2. The fail over algorithm to using a new nim server is described in part 3 of A2, further below.

The basic idea of the fcheck library is that it can be imported and used by code in one node to detect if another node, that is also using the fcheck library, has failed or not. The fcheck library uses a simple heartbeat-ack protocol that resembles the nim protocol you implemented in A1. fcheck is application-independent, and even though you will use it in your nim server, you should build it in a way that allows you to re-use it in A3.

In the fcheck library a node may optionally *monitor* one other node and also allow itself to be monitored by any number of other nodes. In A1, your client will monitor a single nim server (the one the client is currently connected to). Some of the nim servers that the client will connect to will also monitor your nim client. Monitoring means that the monitoring node actively sends *heartbeat* messages (a type of UDP message defined below) to check if the node being monitored has failed, or not. Upon receiving a heartbeat, fcheck must respond to the sender with an *ack* message. Failure is determined/defined based on some number of heartbeats that have not been acked. The exact policy is described below. Note that your fcheck library should respond to a heartbeat message regardless of which node sent the heartbeat (the current nim server or some other server).

The following diagram illustrates an example four node system and the failure monitoring relationships between the nodes with the fcheck (fc) library. In the diagram, Nodes 1 and 3 monitor Node 2 (e.g., Node 1 sends heartbeats and receive acks). Node 2 does not monitor any node. Node 4 monitors Node 3 but is not monitored by any node. Note that a single instance of the fcheck library may monitor zero or one node, and may be monitored by zero or more nodes.



2.2 fcheck API

Your fcheck must provide the following API. Note that below, each call (Start or Stop) invoked by the client must run to completion and return before another invocation by a client can be made. For presentation purposes we list two distinct Start functions below. In the starter Go code, however, there is just one Start function that takes a struct that can be used to determine if the first or second variant of Start is intended.

Note that Start can be invoked multiple times, but Start must be invoked first (before Stop) and calls to Start must alternate with calls to Stop. Below, if `err` (of built-in error type) is nil then the call must have succeeded, otherwise `err` must include a descriptive message of the error. There are no constraints on what the error message is, the exact text does not matter.

- `err` ← **Start**(`AckLocalIP:AckLocalPort`)
  - Starts the failure checking library in a mode where it does not monitor any node, but it does respond to heartbeats (e.g., Node 2 in the diagram above).
    - `AckLocalIP:AckLocalPort` : the library must *receive* heartbeat messages on a *local* UDP `AckLocalIP:AckLocalPort` address. The responses (ack messages) should be always directed to the source LocalIP:LocalPort of the corresponding *received* heartbeat message. The library must use the UDP `AckLocalIP:AckLocalPort` address for sending these heartbeats. Note that fcheck can only receive and respond to heartbeats on a single AckLocalIP:AckLocalPort. Inappropriate ip:port values should result in an error, with the err appropriately set.
- `notify-channel, err` ← **Start**(`AckLocalIP:AckLocalPort`, `epoch-nonce`, `HBeatLocalIP:HBeatLocalPort`, `HBeatRemoteIP:HBeatRemotePort`, `lost-msgs-thresh`)
  - Starts the failure checking library in a mode where it monitors one node and it also responds to heartbeats (e.g., Nodes 1,3,4 in the diagram above). The returned `notify-channel` channel must have capacity of at least 1 and must be used by fcheck to deliver failure notification of the monitored node.
    - `AckLocalIP:AckLocalPort` : see description above.
    - `epoch-nonce` : an epoch nonce value that must be included in every heartbeat message (see below).
    - The library must start monitoring (sending heartbeats to) a node with remote UDP address `HBeatRemoteIP:HBeatRemotePort` using the local UDP address `HBeatLocalIP:HBeatLocalPort`. The `lost-msgs-thresh` specifies the number of consecutive and un-acked heartbeats messages that the library should send before triggering a failure notification.
- `nil` ← **Stop**()
  - Stops the library from monitoring (sending heartbeats) and from responding to heartbeat messages. This call always succeeds.

Notification semantics:

- Monitored node failure notification must be delivered on the `notify-channel` that is returned by Start.
- A failure notification for a node must be represented (on the notification channel) by the structure:

```
type FailureDetected struct {
    UDPIpPort string     // The HBeatRemoteIP:HBeatRemotePort of the failed node.
    Timestamp time.Time  // The time when the failure was detected.
}
```

- If a node X was detected as failed, then (1) exactly one failure notification must be generated, and (2) the library must stop monitoring node X after generating the notification.
- After the call to `Stop()` has returned, no failure notifications must be generated.
- After the call to `Stop()` has returned, heartbeats to the library should not be acknowledged.

2.3 The fcheck protocol on the wire

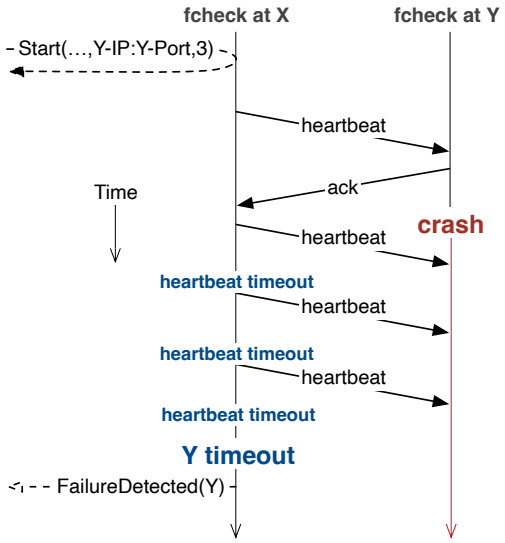
The heartbeat and ack messages in your fcheck implementation must have a specific format:

```
type HBeatMessage struct {
        EpochNonce uint64 // Identifies this fcheck instance/epoch.
        SeqNum     uint64 // Unique for each heartbeat in an epoch.
}
```

```
type AckMessage struct {
        HBEatEpochNonce uint64 // Copy of what was received in the heartbeat.
        HBEatSeqNum     uint64 // Copy of what was received in the heartbeat.
}
```

- EpochNonce is used to distinguish different instances of fcheck. On restart, fcheck would be initialized with a different EpochNonce (with call to Start), and it must ignore all acks that it receives that do not reference this latest EpochNonce.
- The SeqNum is an identifier that is an arbitrary number which uniquely identifies the heartbeat in an epoch. This number could start at any value.
- HBEatEpochNonce and HBEatSeqNum in the AckMessage simply identify the HBeatMessage that the ack corresponds to.

The diagram below is a "time-space" diagram (time flows down) that illustrates how an fcheck instance at node X would detect the failure of node Y (by not observing acks from the fcheck instance at node Y):



Note how fcheck at X resends the heartbeat message (after an RTT timeout) exactly three times (based on the `lost-msgs-thresh` value of 3 passed to Start). After three heartbeat messages have **all** timed-out, fcheck at X times out node Y and generates a failure notification. Your fcheck must implement this behavior precisely.

In general, your timeout mechanism should behave as follows:

- If an ack message is not received in the appropriate RTT timeout interval, then the count of lost msgs should be incremented by 1.
- When an ack message is received (even if it arrives after the RTT timeout), the count of lost msgs must be reset to 0.
- Acks that arrive after a failure notification has been generated (or after `Stop` has been invoked) must be ignored.

2.4 Round-trip time (RTT) estimation

Latency between nodes varies. We want the failure check library to detect failures in networks with a wide range of latencies. We also want to minimize the chance of a false positive (spurious failure detection event).

Your library must wait for a monitored node to reply to a heartbeat with an ack (stop-and-wait protocol). This means that there should, at most, be one heartbeat message in the network from a monitoring node to a monitored node. Only if the node does not reply in RTT time, then should the library send another heartbeat. How long should the library wait for a reply? Depending on where the node is physically located, the wait time will have to vary. Your library must implement a simple RTT estimator to customize the waiting time for the node being monitored. Note that this waiting time may vary for different heartbeat messages.

Your RTT estimator should work as follows:

- Use a value of 3 seconds as the initial RTT value when Start is invoked.
- Each time an ack is received the library should (1) compute the RTT based on when the heartbeat was sent, and (2) update the RTT to be the average of the last RTT value and the computed RTT value in (1). Note that the same calculation must be performed even if the ack message arrives *after* the RTT timeout (but only if it arrives before a failure notification is generated).
- The library should forget the RTT value after Stop is invoked (and use 3 seconds on the next call to Start).

If fcheck receives an ack before the RTT timeout, then when should it send the next heartbeat? Your library should send the next heartbeat no longer than RTT time since the heartbeat that the received ack is acknowledging.

Example scenario. Let's say the first heartbeat was sent at time 1 with RTT value of 5, and an ack for this heartbeat arrived at time 2. Then, the second heartbeat should be sent before time 6 (=5+1). In this example, since the heartbeat was sent at time 1 and the ack was received at time 2, the computed RTT for this heartbeat-ack pair is 1. The new RTT for the connection must be updated as RTT' = (RTT + RTT-measurement) / 2 = (5 + 1) / 2 = 3. When the next heartbeat is sent at time 6 its re-transmission should occur at (6 + RTT) = (6 + 3) = 9.

3.1 Detecting nim server failures and failing over to a new server

Now that you've completed the fcheck library, it is time to integrate it with your nim client. The integration is simple: before your nim client connects to the nim server (sends an opening move), it should begin monitoring the nim server. When fcheck notifies the client that the nim server has failed, the client should *fail over* and use another nim server. That is, it should invoke Stop on fcheck, select a new nim server, invoke Start to monitor the newly selected nim server, and attempt to continue the nim game with the new nim server. The spec in A1 needs to be adjusted in the following ways to support fail over:

1. the client must be aware of multiple nim servers,
2. the client must have an algorithm to select the next nim server,
3. the client must resume the game with the new nim server, and
4. the client must have an algorithm for what to do when all the nim servers are down.
5. the client should trace new actions related to fail over

Multiple nim servers. The client's configuration file will include a list of nim servers the client should use. This list will be ordered and will include at most 8 nim servers.

Selecting the next nim server. The client should use the round robin policy for failing over: it should choose the next server in the list specified in the config file. If it reaches the end of the list, it should continue by selecting the first nim server in the list (that by now might have come back up).

Resuming the game with a new nim server. When failing over to a new nim server, the client should send its last valid move with the appropriate GameState, MoveRow, and MoveCount fields values. The client should not send an opening message to the new nim server if it already started a game with another nim server. However, if no game was started (the client has not previous received a StateMoveMessage from any nim server), then the client should send an opening move message to the server to which it fails over.

When all the nim servers are down. If the client has attempted all the nim servers in the input servers list, and all the servers were reported as failed by the fcheck library, then the client should record the action `AllNimServersDown` and then exit. Note that if the servers list is of length N and the client was interacting with a non-failed server i (i < N), then the client must have (1) detected server i as failed, **as well as** (2) detected all servers j, j>i and j≤N, as failed without any StateMoveMessages from these servers, **as well as** (3) detected all servers j, j<i, as failed without any StateMoveMessages from these servers before recording `AllNimServersDown` and exiting.

Example scenario. Let's say there is 1 client and 3 servers: S1, S2, S3. The client starts the game with S1, makes two moves (records several `ServerMoveReceive` actions). S1 fails (client records `NimServerFailed` action); Client fails over to S2. S2 does not respond (client records `NimServerFailed` action). So the client fails over and tries to use S3. S3 does not respond either (client again records `NimServerFailed` action). At this point, the client has recorded 3 `NimServerFailed` actions after the last `ServerMoveReceive` action. Since the number of servers, N, is 3, at this point the client should record `AllNimServersDown` and exit. The point here is that client should record `AllNimServersDown` only when all servers sequentially did not respond to any move messages. If any one of the servers did respond (and the client recorded a `ServerMoveReceive` action), then the client should keep trying the next server, one at a time round-robin, until the last N did not respond with a move.

New tracing actions. Your A2 solution must trace three new trace actions on the nim client:

- `NewNimServer` : Before starting to monitor a nim server, the client should record the `NewNimServer` action
- `NimServerFailed` : When a nim server has been detected as failed, the client should record the `NimServerFailed` action before initiating fail-over.
- `AllNimServersDown` : If all the nim servers are down, the client should record `AllNimServersDown` as its last recorded action and then exit.

And, as a reminder, your implementation should continue to emit the tracing actions defined in A1:

- `GameStart{Seed}` : marks the start of a new game.
- `ClientMove{GameState, MoveRow, MoveCount}` : indicates that a client has made a move.
- `ServerMoveReceive{GameState, MoveRow, MoveCount}` : indicates that a server's move has been received.
- `GameComplete{Winner}` : marks the end of a game.

However, due to the addition of fail-over, the semantics of some of these actions will need to be redefined. We will describe this next.

Tracing Semantics

Your solution must precisely follow the tracing semantics described below. Your grade depends on the type of tracing log that your solution generates. For example, if traced actions are in the wrong order, or are missing, then you will lose points.

You will use the tracing library as in A1 to report actions using calls to trace.RecordAction. You only need to implement tracing for actions within your own client.

The semantics for [ClientMove] and [ServerMoveReceive] remain the same as in A1. The semantics for [GameStart] and [GameComplete] are slightly updated:

- [GameStart{Seed}] marks the start of a new game.
  - This action should appear before any other Nim-related actions (i.e., [GameComplete], [ClientMove], [ClientMoveReceive], [ServerMove], [ServerMoveReceive]) are recorded. Note that this is different from A1 in which [GameStart] is the first action in a trace. In A2, the first action recorded should be [NewNimServer].
  - Seed must be the randomization seed provided on the command line.
  - Client must record GameStart *at most N times where N is the number of provided nim servers*. For example, if no nim servers fail, then the client will record GameStart just once (as in A1). If one nim server fails, then the client may record GameStart either once (if nim server failed after the game started) or twice (if the nim server failed before the game started). Note that this is different from A1 in which a client's trace should include the GameStart action exactly once.
- [GameComplete{Winner}] marks the end of a game.
  - This action must be recorded *at most* once, after either a legal [ServerMoveReceive] or [ClientMove] (depending on the winner) with all entries in its [GameState] equal to 0.
  - The last recorded action in a client's trace is either GameComplete **or** AllNimServersDown. Note that this is different from A1 in which every client trace was required to terminate with GameComplete.

These semantics above combined with the rules below constitute the tracing rules that your implementation must respect. Below we describe the tracing rules in English. However, we have formal definitions for these in the trace checker. As well, some of these rules depend on tracing actions that are generated by the server (e.g., the server records its own failure as a tracing action before failing).

- The fcheck library used by the client only monitors the nim server that it is currently playing against
- The fcheck library used by the client responds to heartbeat messages sent to it by other nodes
- When the nim server that the client interacts with fails, the client detects the nim server failure (within a time that scales linearly with lost-msg-thresh and is directly proportional to the RTT between client and the nim server).
- When the nim server that the client interacts with fails, the client chooses the correct new nim server: round robin from the supplied list
- Client does not attempt to play with a server that it has detected as failed, unless all other nim servers have been detected as failed
- Client records AllNimServersDown if and only if all nim servers have failed and have been detected as failed by the client.

The formal definitions for fcheck related actions are given below:

- [NewNimServer{NimServerAddress}]
  - This action should be recorded **just before** calling `fcheck.Start`
  - This action should be the first action of the trace
  - For every [NewNimServer] in the trace, there must be a matching [NimServerFailed] or [GameComplete] in the subsequent actions before the next [NewNimServer]
- [NimServerFailed{NimServerAddress}]
  - This action should be recorded **just after** a failure notification is read from the `notify-channel` provided by the fcheck library.
  - This action should be preceded by a [NewNimServer] action
  - The `NimServerAddress` of the [NimServerFailed] and its preceding [NewNimServer] must be identical.
  - For each [NimServerFailed] recorded, there is a corresponding [ServerFailed] with the identical `NimServerAddress` collected in the trace.
- [AllNimServersDown{}]
  - This action should appear at most once
  - This action should not co-exist with [GameComplete]
  - This action must always be the last recorded action if it appears in the trace
  - There must be *exactly* N [NimServerFailed] between the last [ServerMoveReceive] and this action, where N is the length of the nim server list.

Actions server records in the client trace

NOTE: These are the actions recorded by nim servers (and collected by your distributed tracing server). You don't need to record them in your client code, but the traces collected should include them.

- [ServerGameStart]
  - This action is recorded **just after** an opening move is received from a client
- [GameResume]
  - This action is recorded **just after** receving the first message from a client that is not an opening move
- [ServerFailed]
  - This action is recorded by a failing server **just before** it fails
  - A nim server will record this action for every client connecting to it when failing
- [ClientMoveReceive]
  - This action is recorded **just after** a nim server receives the [ClientMove] from a client.
  - For every [ClientMoveReceive] in the trace, there should be at least one [ClientMove] that happens-before it
  - This action and the most recent [ClientMove] that happens-before this action should have the exact same `GameState`, `MoveRow`, and `MoveCount`
- [ServerMove]
  - This action is recorded **just before** a nim server sends a move to a client

Assumptions you can make

- Nim server failure can be detected using the fcheck strategy described above.
- You can assume that all messages fit into 1024 bytes.
- Nodes using fcheck will not monitor themselves (i.e., `AckLocalIP:AckLocalPort` will not equal `HBeatRemoteIP:HBeatRemotePort`).

Assumptions you cannot make

- UDP is reliable.
- Round-trip time between nodes is predictable or bounded.
- *Nim servers do not fail*

Protocol corner cases

- As in A1, during a game, the server will reply with the previous move it made (if it has made such a move) to any invalid message. Note that this means that on fail over, if the first message to a nim server is invalid, the server will not reply to this message.
- Your code should **continue** to use the 1 second timeout for message loss from A1 (before attempting to retransmit a StateMoveMessage message). That is, the RTT computation described above should be used solely in your fcheck library.

Implementation requirements

- **Unlike A1, your client should not implement/include a tracing server. Your client should connect to and use an external tracing server using the information in the config file provided to your client.**
- Your code must be runnable on CS ugrad machines and be compatible with Go version **1.16.7** (which is installed on ugrad machines)
- You must use the UDP message types and tracing action types given out in the starter code.
- Messages must be encoded using `encoding/gob`.
- Your solution can only use standard library Go packages.
- Your solution code must be Gofmt-ed using gofmt.

Solution spec

For fcheck, write a single go source file called `fcheck.go` that implements the fcheck library described above. Download the fcheck.go starter code. Note that you **cannot** change the API in this `fcheck.go`. Our marking scripts will rely on this API to automatically grade your solution. Place your `fcheck.go` file at the `fcheck/fcheck.go` of the UBC GitHub repository that you are using for your submission. But, you can have other files in the repository, e.g., scripts that you have developed.

The client you implemented should act in the protocol described above. You can structure your client program in any way you like, but it must be compilable from the terminal using the following command:
$ make client
The generated executable should be placed at `bin/client`, which can be executed using the command:
$ ./bin/client [seed]
Where [seed] is an arbitrary seed to be sent to the server in the client's first move to generate a game board.

The [config/client_config.json] file must now include several new pieces of information (bolded below). Here is the full listing:

- [ClientAddress]: local address that the client uses to connect to the nim server (i.e., the external IP of the machine the client is running on)
- **[NimServerAddressList]**: an **ordered list of** UDP addresses that specify the nim servers, specifically the UDP ip:port on which each nim server receives new client connections
- [TracingServerAddress]: the address of the tracing server your client should connect to
  y
- [Secret]: ignore
- [TracingIdentity]: always set to "client"
- **[FCheckAckLocalAddr]**: The local address on which fcheck receives heartbeats and on which it sends back acks.
- **[FCheckHbeatLocalAddr]**: The local address that fcheck uses for sending heartbeat messages (and receiving acks).
- **[FCheckLostMsgsThresh]**: the lost messages threshold value to use with fcheck.

Your solution cannot use any external libraries other than those related to the tracing library.

Starter code and testing servers

Download the example fcheck library client code. This code illustrates how a node in a distributed system may use the fcheck library that you are designing. You can (and should) use this client to test your library, though you should also rigorously test your fcheck library with the nim client/servers.

Starter code can be found here.

We will post a set of testing nim server that you can use. There will be four flavours of nim servers:

1. **TracingNimServers** : nim servers that support distributed tracing and do not implement fcheck. Use these in part 1.
2. **TracingFCheckNimServers** : nim servers that support distributed tracing and implement fcheck. These servers can be monitored, but will not monitor you back. Use these in part 2.
3. **TracingFCheckMonitoringNimServers** : nim servers that support distributed tracing and implement fcheck. These servers can be monitored and will also monitor your client. To monitor your node, these servers will assume that your fcheck is responding from a local UDP-IP:(Port+37) that was used to send the server the heartbeats. Use these in part 2.
4. **TracingFCheckFailingNimServers** : nim servers that support distributed tracing and implement fcheck and will fail approximately every 10 seconds. These servers can be monitored and will also monitor your client (as above). Use these in part 3.

The above list of nim servers you can test against will be posted to piazza.

Rough grading scheme

- **Your code must compile and work on ugrad servers**
- **Your code must not change the APIs defined above**
- **You must not change the name and path of your config files**
- **Your code must be configurable using the config files**
- **The command `make client` must be able to compile your client code**
- **The executable of your client must be generated at `bin/client` after compilation**

If any of these are violated, your mark for this assignment is 0. This is true regardless of how many characters had to be changed to make your solution compile, and regardless of how well your solution works with a different API or on a different machine.

The A2 mark breakdown (in terms of the recorded distributed trace):

The actions marked in **blue** are recorded by servers.

- 25% : Distributed tracing works
  - 5% : [ServerGameStart] is recorded after the first [ClientMove]
  - 10% : A [ClientMove] is recorded before each [ClientMoveReceive]
  - 10% : A [ServerMove] is recorded before each [ServerMoveReceive]
- 25% : nim server failures are detected by fcheck in the nim client
  - 10% : If a [NimServerFailed{NimServerAddress}] is recorded, then there is a [NewNimServer{X}] event that happens-before this [NimServerFailed{NimServerAddress}] event and `X` has the same port number as `NimServerAddress`.
  - 15% : Given a nim server with which the client has successfully exchanged a move, if a [NimServerFailed{NimServerAddress}] is recorded, there is a corresponding [ServerFailed{X}] with `X = NimServerAddress` recorded somewhere in the trace.
- 40% : transparent nim server failover works
  - 10% : If there is a [GameComplete] in the trace, then every instance of [NimServerFailed] is followed by an instance of [NewNimServer]
  - 15% : If there is a [GameComplete] in the trace, then every instance of [NimServerFailed] is followed by an instance of [GameResume] or [ServerGameStart] (if the game has not started yet).
  - 15% : If there is a [GameComplete] in the trace, then the game progresses and finishes correctly, according to the rules in A1
- 10% : nim servers total failure handled properly
  - 5% : If there is a [AllNimServersDown] recorded, then (1) it only appears once in the trace, and (2) [GameComplete] does not appear in the trace.
  - 5% : There are exactly N [NimServerFailed] between the last [ServerMoveReceive] (or the start of the trace, if there's no [ServerMoveReceive] in the trace) and [AllNimServersDown], where N is the length of the nim server list.

Advice

- Finish part i before starting on part i+1.
- Test your code with the appropriate testing nim servers in part i before moving on to part i+1.
- Have the fcheck SeqNum start at 0 and increment by 1. This will help with debugging.
- Compile and run your code on the ugrad servers.
- If you write code that uses goroutines, then consider using locks to serialize access to data structures that might need to be modified and accessed by more than one goroutine
- Use gob encode/decode for sending structs across the network.
- The time package is helpful for measuring elapsed time between heartbeats and ACKs.
- You might find the SetReadDeadline method useful.
- Note that due to NATs and Firewalls you may be unable to reach your fcheck clients running on ugrad servers from outside of the UBC network. Either test by running all code on ugrad nodes or by using the UBC VPN.

Make sure to follow the course collaboration policy and refer to the submission instructions that detail how to submit your solution.

Last updated: January 25, 2022