Implementing Exactly-Once Semantic for R2P2

Paul Renauld
Ecole Polytechnique Fédérale de Lausanne
DCSL - Semester Project - Spring 2020

***Abstract*— Remote Procedure Calls in data centers are widely used and their performances are critical. The R2P2 protocol was designed specifically for this purpose, where a request is sent by a client and the server responds with a reply. However, in case of losses of the request or the reply, the guarantees concerning the execution of the request for the application are weak, and it has to choose between executing the request at most once, or at least once.**

We propose an augmentation to R2P2 to give the application the choice of a third, stronger guarantee: exactly once. This ensures that the request will be executed once by keeping local records on the server. The client re-transmits unanswered requests, and the server executes them only if it was not done before.

Scaling is ensured by deleting the records once they are not needed, through the use of acknowledgement messages and timeouts. Performances for the client application are similar to regular R2P2, and the protocol can withstand several failures in links and machines.

I. INTRODUCTION

The structure of data centers is very different than the global internet. Packet losses and link failures are exceptional and machines communicate with nearby machines with low latency and high throughput. For these reasons, regular communication protocols, such as TCP RENO or TCP CUBIC are not suited for data centers.

We focus here on Remote Procedure Calls (RPC), where a client machine needs to perform an action (request) on another machine (server) and gets back a reply. The Request Response Pair Protocol (R2P2 [1]) has been developed for this need. Packet losses, although rare, are still possible, and the application has only limited guarantees that its request was executed. Our goal is to extend this protocol to include the exactly-once semantic.

We will first describe in more detail how R2P2 works and why the exactly-once semantic can be required. We will see previous work done on this subject and then introduce a first implementation for R2P2 before describing the final validated protocol. Finally, we will discuss its validity and possible improvements.

II. BACKGROUND

A. R2P2 Description

R2P2 [1] (Request Response Pair Protocol) was specifically designed for RPCs in data centers which run latency-critical application. It is based on UDP. This avoids the need for a handshake and its overhead that would be present with TCP. It uses only the abstraction of a Request-Reply pair, with no notion of long term client. That is, two requests are treated the same way whether they are from the same client or not. Similarly, the client does not distinguish the servers, thus enabling the implementation of any request-level load balancing. Request/Reply pairs are uniquely identified on the server-side by the tuple `<src_IP,src_port,req_id>` where `src` is the client.

Figure 1 describes the interactions and the packets exchanged for a typical R2P2 request. This is a complete example in the case of several servers with a load balancing router and with requests and replies that need multiple packets. To summarize, the first packet is sent by the client to the router (1) that chooses a server to forward the request (2). Then the server informs the client that it was selected (3) and the client can transmit directly the rest of the request to the server (4). Finally, the server sends back the replies in several packets to the client (6).

Note that when we have a single server, if the requests and replies are short enough to fit in a single packet, only two packets are sent in total for the request and the reply.

R2P2 is implemented in C. The application on the client uses a `struct r2p2_ctx` to setup

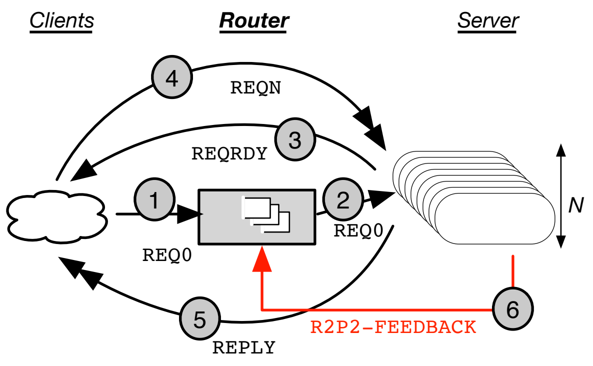


Fig. 1: The R2P2 protocol for a request-reply exchange. Each message is carried within a UDP packet. Single arrows represent a single packet whereas double arrows represent a stream of datagrams. (Source: [1])

several parameters, such as the different callbacks in case of success, failure, and timeout. It then calls a single function `r2p2_send_req` to send a request.

Similarly, the application on the server sets up its call back used when a request is received, and calls a function `r2p2_send_response` to send back the reply.

B. *Need for Exactly-Once semantic*

In its current state, R2P2 provides only two possible guarantees regarding the number of execution of a request:

1) At-most-once: the application does not retry a failed request. It does not know if it was executed or not, but it knows that it was not executed more than once. This can be useful for time-critical applications where a reply is needed either right away, or never. For example, a WebSearch would interrogate many servers and send back an aggregation of the result. If one of the sub-searches fails, it should be ignored to not delay the whole search.
2) At-least-once: the application retries a failed request until it is a success. It then knows that the request was executed successfully at least once, but it could have been executed several times. This is often used for example to write a value in a database. It does not matter to the application if the value is written several times, as long as it is sure that it is written at the end of the request.

These guaranties are often enough, as for the examples described. However, other use cases need to be certain that a request was executed exactly once. For example, a request that increments a counter needs to be executed once, because any other number of execution would lead to different results. This notion is also useful to guaranty atomicity and serializability.

Several protocols exist to implement the exactly-once semantic. We will look at one of them in the next section.

C. *RIFL*

RIFL [2] provides a mechanism for converting at-least-once RPCs to exactly-once. The key core components are briefly described here.

1) Client lease and identification: To distinguish and recognize clients, RIFL proposes the use of a client lease and unique sequence numbers. When a client starts to send requests, it first gets a 64-bit lease number that is put in the request headers as an identifier. The client then sends requests with increasing 64-bit sequence numbers. It means that the requests are uniquely identified by the tuple `<client_id, request_id>`. Note that the large size of the sequence numbers and lease numbers means that the maximum value will reasonably not be reached.

2) Server states: The servers keep a record of each request-pair. When a server receives a request from a client, it checks if it already has a record concerning this request. This leads to four possible situations:

1) `NEW`: the request was not seen before, so the server adds a record for it and starts the computation. Upon completion, it will add the reply to the records and send it back to the client.
2) `IN_PROGRESS`: the request was already received and is being computed. Depending on the implementation, the server could either send a message to the client to signify the computation is still on-going, or it could simply ignore the request.
3) `COMPLETED`: the request was already received and computed, and the reply was sent back. The server sends back again the reply.

4) `STALE`: the request was already received and the reply was sent, but the record has been garbage collected (see next section) and this request should be ignored. Depending on the implementation, the server could send a message to the client to signify it. This happens when a request is significantly delayed in the network.

3) Garbage collection: To scale, RIFL needs to garbage collect records as requests arrive. This is done with two mechanisms.

First, the client's lease can expire if it is not renewed regularly. When a lease expires, all the servers can destroy the records concerning this client. The lease number will never be used again.

Second, the request's header contains the field `firstIncomplete`. This is the lowest request ID for which the client did not receive a reply. This works as an acknowledgement to the server. That is, when the server receives a request, it can delete all the records of this client with a sequence number lower than `firstIncomplete`.

To avoid overwhelming the servers, the client will not emit too many requests at once. It keeps a window of on-going requests that can go up to 512 in the original paper, but this value can be changed. Formally, this means that for any request, this always holds: $\texttt{firstIncomplete} + 512 > \texttt{request\_id}$.

4) Failure: If a client does not get a reply for its request, it proceeds as in the at-least-once semantic. That is, it re-sends the request with the same sequence number after a timeout.

Note that if no packet is dropped and requests/replies can fit in a single packet, only two packets are exchanged per RPCs, plus the overhead of the potential lease renewal.

III. Requirements

We will implement the exactly-once semantic on R2P2 with the following requirements.

A. Functionality

When activated, the exactly-once semantic should provide guarantees that no request will be executed twice, and the best effort will be done to always execute it once, even in case of several packet losses. In some very rare cases, it is possible that the protocol fails, i.e. the request may or may not have been executed, and the client should check by some other means if it was correctly executed. This semantic should be usable in a system that uses regular R2P2 at the same time.

B. Semantic

The implementation should respect the semantic of R2P2. That is, request-reply pairs should still be present and no state should be kept between the requests. Requests from the same client are not handled differently than requests from different clients.

C. Interface

The change in the interface should be minimal, and the applications should be able to choose between exactly-once or regular R2P2. This means that no field should be added to the `struct r2p2_context` that the client uses to share its intents with the protocol.

D. Performance

Since we are dealing with a system where packet losses are rare, it is important to consider the performances in the best cases scenario. The delay for the client to receive the reply should be similar to regular R2P2 and we should minimize the memory pressure for clients and servers.

E. Assumptions

To simplify at first the system, we will assume that we have only one client and one server and that the server will not crash. We also assume that the requests and replies can fit in a single R2P2 packet. The client can however crash, and any packet drop or delay is possible. Section VI will discuss how the system would hold without some of these assumptions.

IV. First Version

Our first implementation was based on RIFL (section II-C) adapted to R2P2.

A. Concept

The main difference with RIFL is that we won't use a lease management system. Instead, since R2P2 uniquely identifies requests with the tuple `<src_IP,src_port,req_id>`, we will identify the clients with the pair `<src_IP,src_port>`.

This means that we will not garbage collect records for clients with expired leases. Instead, if needed, we could delete records of a client when no request from this client has been received in a long time, but this won't be done here.

We will be using the 16-bit R2P2 request ID instead of the RIFL 64-bit sequence number. We need it to be sequential, and not random as R2P2 is doing. Since a 16-bit unique IDs would be highly limiting, we allows it to wrap around, making the maximum number of requests unlimited. Determining if a received request is a valid new request or an old delayed one is more tricky. We will use the fact that clients have a limited number, say N, of possible outgoing requests at the same time. When a request is received by the server, we compute how far its ID is to the previous `firstIncomplete` value on the server. The request can only be valid if its ID is less than $2*N$ above the `firstIncomplete`.

This requires some additional assumptions. We are assuming that we will not have different requests with the same ID at the same time in the system. This is partly ensured by the upper bound on on-going requests on the client, but packets could still be delayed. Therefore, we are assuming that the system will not reuse the same request ID before all copies of the previous request with this ID are gone from the network.

B. *Implementation*

1) Interface: To satisfy the interface requirement (III-C), the only modification to the public API is the addition of the function:

```
void use_exct_once(struct r2p2_ctx *ctx);
```

It should be called after all the values in the context have been setup by the application. To store in the context whether the exactly-once semantic should be used, we will use the 3rd bit of the type policy, which was previously unused and always 0.

2) Communication: Two messages types are added: `REQUEST_EXCT_ONCE` and `RESPONSE_EXCT_ONCE` for requests and replies respectively.

3) Client side: The client needs to keep the state of a sequence number to ensure that requests are in sequential order. It also needs to track which requests were sent and did not receive a reply yet as well as the lowest unacknowledged sequence number. The states of the client are shown on listing 1. When the client sends a message with exactly-once activated, the request ID will be `next_seq`, and `next_seq` will be incremented. A bit corresponding to the request is set in `req_acks` which keeps track of on-going requests that were not answered yet. The request will instantly fail if there are already too many pending requests.

```
struct r2p2_eo_client_info {
    uint16_t next_seq;
    uint16_t lowest_pending_ack;
    bool req_acks[EO_WIN_SIZE];
};
```

Listing 1: Client-side states (first version)

When sending a request, the client also needs to include in the header the `lowest_pending_ack`. This means that the header size is now dynamic and will be 16 bits longer for exactly-once requests only. This is enabled by the field `header_size` in the R2P2 header [1]. This requires to modify some of the existing code: `sizeof(struct r2p2_header)` cannot be used directly, the size of the header should be looked up every time.

When the client receives a reply for a request that was previously unanswered, it forwards it to the application and updates `req_acks`. Doing so, the `lowest_pending_ack` could also be updated. It will be the ID of the lowest unanswered request, i.e. the lowest `true` value in `req_acks`.

In case of a timeout, the client resends the requests.

4) Server side: The states for the server are shown in listing 2. Each thread keeps a collection of `struct r2p2_eo_server_window`. For each of its clients, the server has records of received requests and their replies.

When receiving a request, the server needs to determine which of the four RIFL states (described in II-C.2) applies:

1) `NEW` applies when no record exists for this request number (and client), but the request number is a valid new one, given `window_start`. In this case, the computation is started and a record is created.

```
struct r2p2_eo_record {
  int state;
  struct r2p2_msg *reply;
};

struct r2p2_eo_server_window {
  struct r2p2_host_tuple client;
  uint16_t window_start;
  struct r2p2_eo_record
    records[EO_WIN_SIZE];
};
```

Listing 2: Server-side states (first version)

2) IN_PROGRESS applies if a record exists for this request, but the reply is not available, because the computation is still on-going.
3) COMPLETED applies if the record exists with a reply. The reply is sent again back to the client.
4) STALE applies if no record for this request exists, and the request number is not a valid new one.

The server also needs to handle the ack when receiving a request. It advances window_start up to the least unacknowledged request number, and clears all the records below this.

When the application emits a reply to a request, the corresponding record is updated to include the reply, and its state is now COMPLETED.

C. *Limitations*

As described, the server keeps a state for each client. Requests coming from the same client are not independent because of the acknowledgement mechanism. This violates in part the semantic of R2P2 and requirements described in III-B. Thus, this implementation and the use of RIFL is not desirable, and an alternative solution should be found.

V. Final Version

In this section, we describe the final protocol that has been implemented.

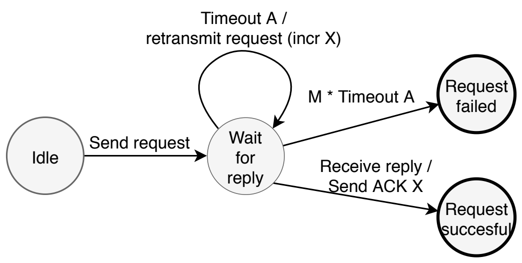
A. *System Description*

The new best-effort implementation keeps the semantic of R2P2. The state machine of the client is presented in Figure 2a while the server's one is in Figure 2b. Significations of the parameters can be found in Table I. The idea is that a request is executed as with regular R2P2, but the server keeps track of the requests it recently executed and the client sends back an acknowledgement when receiving the reply.

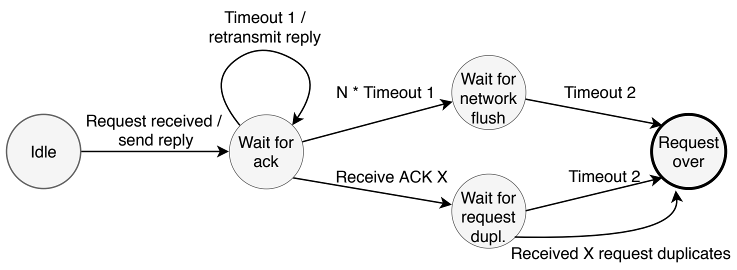
In the usual case where no packets are dropped, three packets (request, reply, ack) are sent, the delay for the client to get the reply is unchanged, and the used memory is released as soon as the operation ends. The common execution goes as follows:

1) The client sends the request and creates in-memory records for it.
2) The server receives the request, creates in-memory records for it, computes the result and sends it back to the client.
3) The client receives the reply, sends back the ACK 0, transmits the reply to the application and deletes its local record.
4) The server receives ACK 0 and deletes its records of this request.

When the client does not receive a response after a timeout, it re-transmits the request. Similarly, if



(a) Client-side logic



(b) Server-side logic

Syntax: <event>/<action>. All in-memory states are deleted in the final states, indicated by round black borders

Fig. 2: Final implementation logic

Name	In-code macro	Signification	Lower bound
Timeout A	`EO_TO_REQUEST`	Timeout between retries to send the request	$>$ RTT $+$ Compute time
M	`EO_MAX_RETRY_REQUEST`	Maximum number of retries to send a request	/
Timeout 1	`EO_TO_REPLY`	Timeout between retries to send the reply	$>$ RTT
N	`EO_MAX_RETRY_REPLY`	Maximum number of retries to send a reply	s.t. $N *$ Timeout 1 $> M *$ Timeout A
Timeout 2	`EO_TO_NETWORK_FLUSH`	Maximum lifetime of a packet in the network	Defined by the network

TABLE I: Protocol parameters

the server does not receive the ack, it re-transmits the reply. The client can delete all its records once it receives the reply. For the server, it needs to be sure that it will not receive a re-transmission of the request before freeing its states. This happens either when:

- The server received all the duplicate requests sent by the client. To know this, the client includes the number of re-transmitted requests in the ack (`ACK N`).
- All the requests either arrived at the server or were dropped by the network and will never be received. This introduces an assumption on the maximum survival time of a packet in the network: `Timeout 2`

Note that an ack is never duplicated. When the client sends an ack, it deletes all sates for this request. If a duplicated reply arrives later, it is simply ignored. This means that, in the rare occasions that the ack is lost, the server will re-transmit the reply up to `N` times, before deleting the states. This is transparent to the client and has a small impact on the server that needs to keep states in memory for longer.

Note also that the acks are not necessary for the correctness of the protocol, but are used to lower the memory load on the server.

Requests are identified with the tuple `<src_IP, src_port, req_id>` as in regular R2P2.

B. Assumptions

Table I shows assumptions for lower bounds of the protocol parameters. Most parameters can be changed arbitrarily as long as they are consistent together. However, the Timeout 2 is directly defined by the network, and it corresponds to the maximum delay between sending a packet and the point this packet will never be received by the server if it was not received already. Thus, we need to make the assumption that such a delay exists that the value could be known for the network.

The requests are distinguished using the 16-bits request ID. Once all values have been used, the ID wraps around and goes back to 0. Given that no two different requests with the same ID can exist in the system at the same time, we need to assume that the minimum delay between two usages of the same ID is greater than the maximum lifetime of a request. This maximum lifetime is $M*$Timeout A$+$RTT$+N*$Timeout 1$+$Timeout 2, i.e. it happens when all but the last requests are lost, then all but the last replies are lost, then the ack is lost.

C. Implementation

1) Interface and communication: The interface is the same than for the first version described in IV-B.1. The message types `REQUEST_EXCT_ONCE` and `RESPONSE_EXCT_ONCE` are also used, but a third one is added for the ack messages: `ACK_EXCT_ONCE`. Note that this ack is not related to the ack messages used by R2P2 when using a router [1], i.e. `ACK_MSG`.

2) Records: The current states of R2P2, `r2p2_server_pair` and `r2p2_client_pair` are simply extended to accommodate the exactly-once semantic. This means that the semantic of R2P2 is respected, and requests are independent. The extension is accomplished by adding a pointer in the current structures to additional states. When this pointer is `NULL`, this means that exactly-once is not used for this particular request. This also enables us to use the exactly-once semantic for some requests while keeping regular R2P2 for the others.

The additional states for the server are described in listing 3. Note that, unlike the `client_pair`, the `server_pair` does not have an associated timer, so it is added to the additional states. The other variables are respectively to keep track of the number of received requests, the number of re-transmitted requests as indicated by the ack message (can also hold the value ACK_NOT_RECEIVED to indicate that the ack was not received yet), and the number of re-transmitted replies.

```
struct r2p2_sp_exct_once_info {
  uint16_t req_received;
  uint16_t req_resent;
  uint16_t reply_resent;
  void *timer;
};
```

Listing 3: Server-side states (final version)

The additional states for the client side are shown in listing 4. It simply keeps track of the number of re-transmitted requests.

```
struct r2p2_cp_exct_once_info {
  uint16_t req_resent;
};
```

Listing 4: Client-side states (final version)

3) Client-side:

a) Request sending: Sending a request with exactly-once is almost the same than for regular R2P2. The only difference is that the states must be allocated with the additional exactly-once states and initialized accordingly. The request is the same except for the request type, and the timer will be started as usual.

b) Timer triggered: When a timer is triggered, the client resends the same request, increments req_resent and restart the timer. Once req_resent reaches its maximum value (EO_MAX_RETRY_REQUEST), the request fails, the corresponding callback to the application client is used, and the states are deleted.

c) Reply received: Before executing the regular behavior of R2P2 when receiving the reply (disarm the timer, callback the application, delete the states), the client sends back the ack message. It is similar to a regular R2P2 message, but its type is ACK_EXCT_ONCE, the length is always 2 bytes and the content is req_resent, i.e. the number of times the request was re-transmitted (so 0 in the optimistic case).

When a reply is received for a request that is not available in the client's states (i.e. the client already discarded it because it timed out or it received the reply already), it is simply discarded by R2P2.

4) Server-side:

a) Request received: When a request is received for the first time by the server, a server_pair is allocated with the exactly-once states and initialized accordingly. A timer is also allocated, as described in V-C.5. Unlike regular R2P2, the pair is systematically added to the collection.

b) Reply sending: When sending the reply, the corresponding pair is not de-allocated, but it is still in the states of the server. The timer is started for the duration EO_TO_REPLY.

When a duplicate of the request is received, req_received is updated and the duplicate is discarded. The server then tries to garbage collect the request pair. That is, if the ack was received as well as all the duplicates of the request (as indicated in the ack), the states of this request are removed.

c) Ack received: When an ack is received, the server updates req_resent and tries to garbage collect the states of the request.

d) Timeout: The first kind of timeout is while waiting for an ack when the reply was re-transmitted less than EO_MAX_RETRY_REPLY times. In this case, the duration was EO_TO_REPLY. The reply is re-transmitted and the timer is restarted. The other kind of timeout is when we are waiting for the network to flush any potential duplicate of the request. In this case, the duration was EO_TO_NETWORK_FLUSH. The server can remove all the states corresponding to this request.

5) Timers: Timers for the client-side were already implemented in R2P2. Here we needed also timers for the server-side. This is implementation-specific, and it was implemented for Linux in this project, using the sys/timerfd.h library and the epoll system. This is done by introducing the timer pool described in listing 5.

Allocating a new timer is done similarly to the sockets. We have a pool of timers that we go through until we find one that is available.

When the server is initialized, it creates the timers with timerfd_create and registers them with epoll_ctl.

```c
struct loose_timer {
  int fd;
  int taken;
  void *data;
};

struct timer_pool {
  uint32_t count;
  uint32_t idx;
  struct loose_timer
    timers[TIMERPOOL_SIZE];
};
```

Listing 5: Timers on Linux implementation

When an event for one of these timers occurs, `sp_timer_triggered` is called for the server pair contained in `data`.

All the additions to the internal API that are implementation-specific are shown in listing 6. Their roles are straight forward.

```c
int cp_restart_timer(
    struct r2p2_client_pair *cp,
    long timeout);
int sp_restart_timer(
    struct r2p2_server_pair *sp,
    long timeout);
void sp_get_timer(
    struct r2p2_server_pair *sp);
void sp_free_timer(
    struct r2p2_server_pair *sp);
```

Listing 6: Implementation-specific functions for timers

Note that, except for the callback, these `loose_timers` are not dependent on the server pairs. They could be easily used for another purpose later on in R2P2.

D. Evaluation

In this section, we will look at several execution scenarios, with possible failures or issues with the links and the client. Each scenario has a unique failure, but the behavior of the protocol with several failures can be inferred. We will present the observed execution, which is always the same that what would be expected from the description of the protocol.

Tests were run with a single client and a single server, on the same machine. We developed a bash script, using the `tc` Linux tool, to easily add delays for packets either going in the server's port, coming out of it, or both. A delay of 1s was systematically added in both directions to be able to observe the execution in real-time and to create failures during the execution. To simulate a broken link, we raised the delay to 1000s, while a slow link had a delay of 4s. The computation time of the request by the server is negligible.

The parameters (Table I) were:

$$\text{Timeout}A = \text{Timeout}1 = 2.5s$$
$$\text{Timeout}2 = 5s$$
$$N = M = 5s$$

Observations were made by dumping the states of the records on both the server and the client at various points in time, and by analyzing the network traffic with Wireshark.

1) Normal execution: Here no link is broken and no machine crashes. As intended, we see one packet going out of the client, then the server creates a record for it and sends back the reply, then the client sends back the the ACK0 and deletes its record, finally the server deletes its own record.

2) Broken or slow links:

a) Broken link during the whole execution: The client tries to send the request and times out every `TimeoutA` and eventually fails after M retries. The server never receives anything. From the client's perspective, this is equivalent to communicating with a server down.

b) Initially broken link: The client sends the requests `X` times until the link is back up. The server receives the request and transmits the reply to which the client answers with `ACKX`. The server receives it and keeps its record until `Timeout2` expires.

c) Initially slow link: The client sends multiple requests that are delayed. The server eventually receives one and sends back the reply. The client gets it and sends the ack (on a link that is not delayed anymore). The server gets the ack, waits to receive all the requests, and deletes its record.

d) Broken link after transmission of the request: The client sends the request and then the link breaks. The server sends back the reply. The client re-transmits the reply `N` times every `TimeoutA` and then abandon and deletes its record. Simultaneously, the server re-transmits the reply `M` times every `Timeout1`, abandons, waits for `Timeout2`, and deletes its record. Note that the client abandoned before the server.

e) Lost ack/broken link to the server after transmission of the request: The client sends the request and the server sends back the reply. The client then sends the ack message and deletes its record, but the ack is lost. The server re-transmits the reply `M` times every `Timeout1` but the client ignores it, as it does not have any record for this request anymore. Eventually, the server abandons, waits for `Timeout2`, and deletes its record.

3) Client crashes:

a) Client crashes before sending the request: No packet is transmitted, nothing happens.

b) Client crashes after transmitting the request: The client sends the request and crashes. The server gets the request, sends back the reply, re-transmits it `M` times every `Timeout1`, abandons, waits for `Timeout2`, and deletes its record.

c) Client crashes after sending the ack message: The normal execution happens (V-D.1).

4) Long-running request: In this scenario, we look at what happens when the request to execute is significantly longer than expected. To simulate it, we add a sleeping instruction for 5s in the application code on the server-side.

The client sends the request. The server receives it and starts processing it. The client times out and re-transmits the request `X` times. The server receives them, keeps in memory the number of received requests, but otherwise ignores them. The task completes and the server sends the reply. The client receives it, sends `ACKX` and deletes its record. The servers receives `ACKX` and immediately deletes its record as it already received all the duplicates of the reply.

5) Takeaways: We showed that the system can support multiple types of failure according to the requirements. The system still works if some parameters were badly estimated, like the Round Trip Time (shown with slow links) or the computation time, but this leads to a higher cost, with more packet exchanged. However, `Timeout2` still needs to be correctly estimated (or overestimated) to ensure correctness.

VI. Discussion

In this section, we discuss how the protocol holds without some of the assumptions that were made, or what needs to be added for it to hold. We also look at possible improvements that could be made.

A. Client crashes

If a client crashes, restarts, and sends a request with the same ID than a previous request that was still on-going, an issue might occur. This can be solved by keeping long-term persistent states by the client, for example on its hard-drive. Note that this is not an issue if the delay to restart the client is greater than the maximum lifetime of a request.

B. Server crashes

If the server crashes, the persistency of the requests that were validated depends on the underlying application. If the request is executed but the reply does not get to the client, this is the same than if all the replies were lost, i.e. the protocol failed and the client has to verify by some other means if the request was executed. Overall, the protocols can sustain server crashes, as long as the application is adapted.

C. Multiple servers and multiple clients

Having several clients is not an issue, as all requests are independent from the server's perspective. As for multiple servers, the protocol holds as long as all packets with the same tuple identifier, i.e. concerning the same request, are routed to the same server. This is often required by the application, and loadbalancers usually provide this guarantee, so the protocol can easily hold.

D. No maximum packet lifetime

The first assumption of V-B requires that there exists an upper bound to the maximum delay a packet can experience. If this is too strong for a particular network, we can create this property if machines are synchronized with a shared clock. In this case, we can add to the header the timestamp at which it was sent, and the destination should drop the packets for which the sending time and receiving time is greater than a threshold. This threshold is now the artificial upper bound to a packet lifetime, and our protocol can function normally with it.

E. Longer Request ID

If the second assumption of V-B (the minimum delay between two usages of the same ID is greater than the maximum lifetime of a request) is too strong for our system, we can relax it by augmenting the `seq_id`. That is, instead of using a single 16-bit integer, we can add another 16-bit integer to the header, so that the ID of the request is now a 32-bit integer. We still need to have a similar assumption, but the delay between two usages of the same ID is significantly increased ($\times 2^{16}$).

F. Multi-Packet Requests

R2P2 can handle requests and replies that don't fit in a single packet. However, even though it could in theory, the current implementation of R2P2 does not support lost or out-of-order packets for multi-packet requests/replies. We argue that if it did, the exactly-once protocol would hold as well.

For that, the requests and replies would be considered to be received only when all the packets are received.

It is then more likely that the server would need to wait for `Timeout2` to clear its record, as a single packet drop in any of the requests would make the server consider that the request was not received. To mitigate that, the server could delete its record as soon as it received the first packet of each copy of the request. This means that drops of other packets in the requests would not prevent the server from deleting the record early. This way, even though some other packets could be delayed, the whole request could not be received and executed again.

G. Possible optimizations

1) Memory Optimization: To reduce the memory load of the server, a simple optimization would be to delete the reply in the record once it has been acknowledged. Remember that after receiving the ack message, the server might keep the record in memory for longer if at least one duplicate of the request was sent. However, at that time, the content of the reply is not relevant anymore for the server, and it could safely de-allocate it. This would not compromise the correctness of the protocol, but could save memory in rare occasions, especially if the replies are long.

2) Resilience to lost Acks: Another optimization for the server is possible with a tradeoff. Note that when the acknowledgement is lost, the client deletes its record and the server will only deletes its own after a timeout. One could argue that the client could keep in memory its record for some time after sending the acknowledgement so that it could re-send the ack if the client receives a duplicate of the reply. In this case, the server could delete its record when receiving the re-transmission of the acknowledgement, instead of waiting for a timeout. This would effectively save memory for the server, as well as avoiding useless network traffic with reply re-transmissions. This is then a tradeoff between systematically keeping the record on the client-side for slightly longer, or keeping the record on the server-side when the acknowledgement is lost for significantly longer. Given that this protocol is to be used in a network where packet losses are rare, this optimization is unprofitable most of the time.

3) Aggregated Acks: In the normal case, the client sends two packets compared to one for regular R2P2. This can have a high cost, especially when many requests are on-going. Note also that the ack messages are very small, and only contain one number. We could, therefore, aggregate the ack messages together when many of them are sent by the client. This means that the client would not send the ack right away, but would wait a small delay to send it together with other acks. The new aggregated ack would still be a single packet, but it would be different from the other R2P2 messages, as it would not have a single request id anymore.

VII. Conclusion

We presented the protocol R2P2, specifically designed for RPCs, and its need for the exactly-once semantic. After a first implementation relying on RIFL but that did not respect the semantic of R2P2, we developed a working protocol that keeps requests independent while minimizing the cost for both client and server. This protocol was tested and validated for several possible scenarios, and possible improvements or optimizations were discussed.

With this protocol, we are now able to guarantee to the applications that their requests will be executed exactly once by the server, if they require it with a simple interface.

VIII. Acknowledgement

I would like to thank the Data Center System Laboratory of EPFL for allowing me to perform this project. In particular, I am grateful for the help, advice, guidance, and recommendations of Marios Kogias.

References

[1] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX 19)*, pages 863–880, 2019.

[2] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 71–86, 2015.