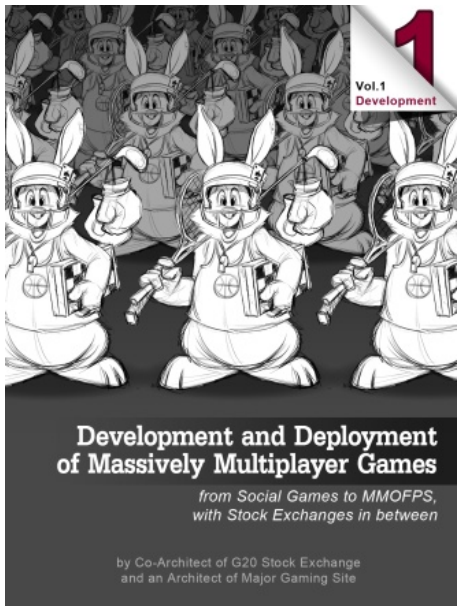
MMOG. Point-to-Point Communications and non-blocking RPCs

posted February 8, 2016 by "No Bugs" Hare, translated by Sergey Ignatchenko

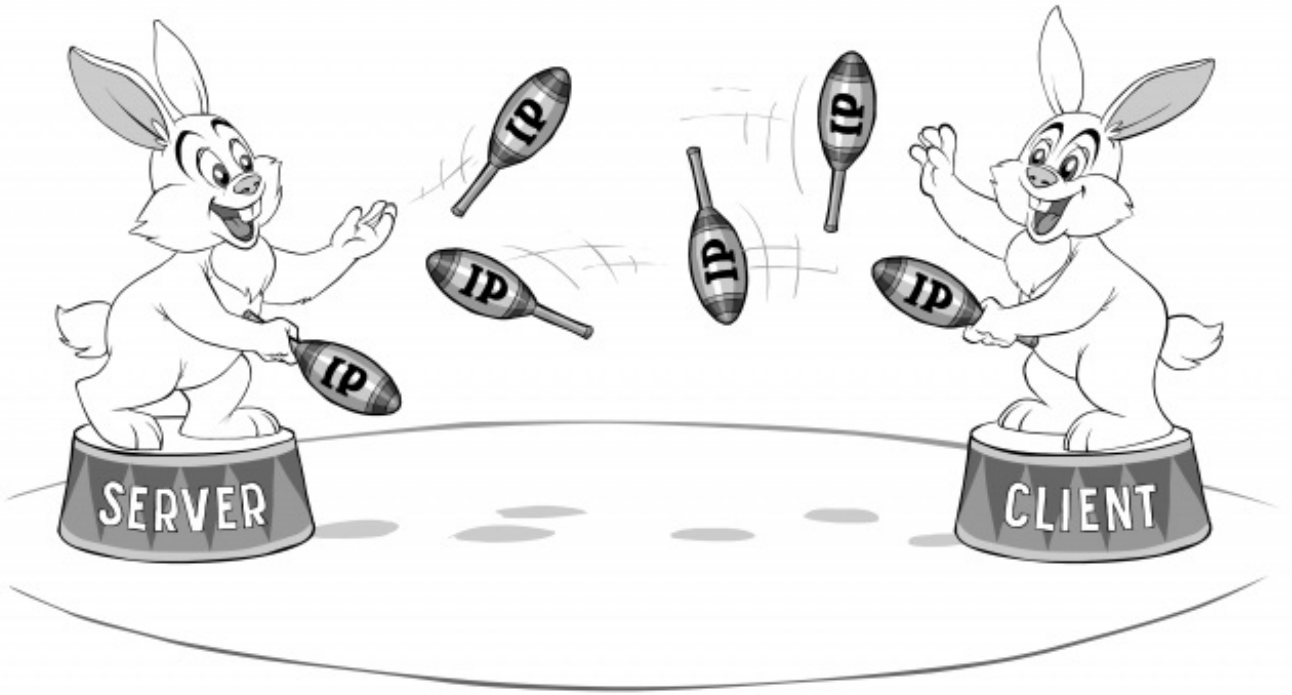
[[*This is Chapter VII(c) from the upcoming book "Development&Deployment of Massively Multiplayer Online Games", which is currently being beta-tested. Beta-testing is intended to improve the quality of the book, and provides free e-copy of the "release" book to those who help with improving; for further details see "Book Beta Testing". All the content published during Beta Testing, is subject to change before the book is published.*

To navigate through the book, you may want to use Development&Deployment of MMOG: Table of Contents.]]



After we've discussed Publishable State, the next thing we'll need for our MMO is Point-to-Point communications. While Publishable State is mostly about servers communicating with clients, Point-to-Point communications can happen either between client and server, or between two servers. These two types of Point-to-Point communications have quite a bit in common, but there are also substantial differences.

Note that differences between TCP and UDP are still beyond the scope until Chapter [[TODO]]; for now we're speaking of *what we need,* and not about *how to implement it.*



RPCs

Regardless of the nature of Point-to-Point communications (whether it's being between client and server, or between two servers), they share certain common properties.

In particular, it is common for games to implement point-to-point communications as non-blocking Remote Procedure Calls (RPCs). While this is not required (and you can use simple message exchange instead – with either hand-written or IDL-based marshalling), non-blocking RPCs tend to speed up development significantly.

It should be noted, however, that while non-blocking RPC are perfectly viable for games, you Really SHOULD keep away from blocking RPC (as in DCE RPC/COM/CORBA)

The reason for it is the following. With games, you SHOULD use event-driven/FSM programming (if I didn't manage to convince you about it in Chapter V, just trust most of game developers out there, and take a note of most of them using FSMs at least to some extent; in particular, classical game loop and simulation loop are FSMs). And with event-driven FSMs, any blocking operation (especially the one which involves waiting for remote entity) is a Big No-No.

Implementing Non-Blocking RPCs

To implement non-blocking RPCs, you need a way to specify signatures of your remotely-callable functions; such specification defines the interface (and often protocol, though see more on encodings in [[TODO!]] section below) between RPC caller and RPC callee. Sometimes (like in Unity), it is done by adding certain attributes ([RPC]/[ClientRpc]/[Command] method attributes in Unity) to existing functions/methods.

However, usually I prefer to have my own explicit IDL (with an IDL compiler) instead. The reason for this preference for a separate IDL is that whenever we specify RPC signatures right in the code, it means that having them in the code-written-in-a-different-language, we'll need at least to specify them once again in the second language (what makes code maintenance extremely error-prone).[1]

We'll discuss implementation of your own IDL in the [[TODO]] section, but for the purposes of our current discussion it doesn't really matter whether we're using intra-language RPC specifications (like in Unity), or our own external IDL (as we'll discuss below).

IDL

An interface description language or interface definition language (IDL), is a specification language used to describe a software component's application programming interface (API).

— Wikipedia —

---

[1] in theory, you could use one language as an IDL for another one, but I haven't seen such things (yet?)

Specifics of Non-blocking RPCs

Non-blocking RPCs have some peculiarities, both for implementing them, and for using them. In general, there are two cases for non-blocking RPCs.

The first case is a non-blocking RPC, which returns void (and can't throw any exceptions). For such void RPCs, everything is simple – caller just marshals RPC parameters, and sends a message to the callee, and the callee unmarshals it and executes RPC call, that's about it. From all the points of view (except for pure syntax), calling such an RPC is the same as sending a message (with all the differences being of purely syntactic nature).

A typical example of such an RPC (as defined in an IDL) is something along the following lines:

```
STRUCT Input {
  bool left;
```

```
  bool top;
  bool right;
  bool bottom;
  bool shift;
  bool ctrl;
};

void move_me(Input in);
```

Non-void RPCs

The second (and much more complicated) case for RPCs is an RPC which either returns a value, or is allowed to throw an exception (or both). An example IDL for such a non-void RPC is the one from Chapter VI:

```
int dbGetAccountBalance(int user_id);
```

"Non-void RPCs are significantly more complicated to implement, and most of the popular game engines out there do NOT support them

These non-void RPCs are significantly more complicated to implement, and most of the popular game engines out there do NOT support them (see Chapter [[TODO]] for more information about Unity/Photon and Unreal Engine).

The main issue with implementing non-void RPCs is for the caller to specify what to do when the function returns (or throws an exception). There are many ways of doing it, they were discussed in Chapter VI, section "Asynchronous Processing for Finite State Machines/Actors: from plain event processing to Futures (with OO and Lambda Call Pyramids in between)" (with FSMFutures being my personal favorite at the moment). On the other hand, while implementing them is difficult, once they are available, they do simplify development significantly, so you will want to use them if your engine supports them.

Whenever your engine doesn't support non-void-RPCs, you'll usually need to make another RPC call in the opposite direction when you're done

In this case, our last example will need to be rewritten along the following lines:

```
//Game World Server to DB Server:
void dbGetAccountBalance(FSMID where_to_reply, int user_id);

//DB Server to Game World Server:
```

```
void gameWorldGotAccountBalance(int user_id, int balance);
```

or in more general manner:

```
//Game World Server to DB Server:
void dbGetAccountBalance(FSMID where_to_reply, int request_id, int user_id);

//DB Server to Game World Server:
void gameWorldGotAccountBalance(int request_id, int balance);
```

While this will work, it is quite cumbersome and inconvenient (substantially worse than even Take 2 from Chapter VI).[[TODO! add these RPC to Chapter VI as "Take 1a"]]

Same-thread operation

> "In other words, you can write your code 'as if' all-your-code-within-the-same-FSM executed within the same thread

Another thing to understand about non-blocking RPCs is that due to non-blocking nature, other things can happen within the same FSM while the RPC is executed. This can be seen as either blessing (as it allows for essentially parallel execution while staying away from any thread synchronization), or a curse (as it complicates understanding), but needs to be kept in mind at all the times you are dealing with non-blocking RPCs. One positive thing to note in this regard is that for most sane implementations, and regardless of using any of the ways to report back described in Chapter VI, you don't need to care about thread synchronization (as all the callbacks/lambdas/futures will be called in the context of the same thread). In other words, you can write your code "as if" all-your-code-within-the-same-FSM executed within the same thread (and whether it will be actually the same thread or not, is not that important); from a bit different perspective, you can think of all the callbacks "as if" they're essentially the same as co-routines (but using a different syntax).

Client-to-Server and Server-to-Client Point-to-Point communications

Now, as we've discussed the similarities between point-to-point communications, we need to describe differences. And arguably the most important difference between Client-to-Server and Server-to-Server communications, is related to disconnects. As a rule of thumb, for Server-to-Server communications the disconnects are extremely rare, and all the disconnects are transient (that is, unless your whole site is down). It means that we can expect that they are restored really quickly, which in turn means that we can try to hide temporary loss of connectivity from application layer. On the other hand, for Client-to-Server (and

Server-to-Client) communications, this "restored really quickly" observation doesn't stand, and dealing with disconnects becomes an important part of application logic.

Let's speak about Client-to-Server and Server-to-Client communications first.

Inputs

One thing which you'll inevitably need to transfer from client to server, is player inputs. For a non-simulation game (think blackjack, stock exchange, or social game), everything is simple: you've got an input – you're sending it to the server right away.

For simulation games, however, it is not that trivial. Traditionally, simulation-based games usually operate in terms of "simulation ticks", and usually single-player games are just polling the state of keyboard/mouse/controller on each tick. As a result, when moving from a single-player simulation game to the network one, it is rather common to mimic this behaviour just by client sending state of (keyboard+mouse+controller) to the server on each tick (which becomes a "network tick"). An alternative (also pretty common) approach would send only *changes* to this (keyboard+mouse+controller) state; this can be done either as soon as the state is changed, or again on "tick".

"However, as soon as we realize that packets can be lost, handling inputs becomes a bit different.

As long as there are no disconnects (nor packet loss), there is no that much difference between these approaches. However, as soon as we realize that packets can be lost, handling inputs becomes a bit different.

If we're transferring state of player's input devices on each tick, then in case of lost packet[2] PC will effectively stop on the server-side; moreover, at the same time, if we implement Client-Side Prediction, it will be running on the client side.

On the other hand, if we're transferring only *changes* to keyboard/mouse/controller state, then in case of packets being lost, our PC will keep running for some time (until we detect disconnect) even if player has already released the button; this may potentially lead to PC running off the cliff even if the player's actions didn't cause it ☹ (just by disconnect happened at an unfortunate time).

A kind of "hybrid" approach is possible if we're using client-to-server acknowledgment packets (which will arise in a pretty much any game world state publishing schema, see Chapter [[TODO]] for further discussion) to distinguish between "player is still keeping the button pressed" and "we have no idea, as the

packet got lost" situations. In other words, if an acknowledgment arrived, but without any information about the keyboard state change – then we know *for sure* what is going on on the client side,[3] if there is no acknowledgment – then something is wrong, so our server can stop PC before he runs off the cliff.

Overall, there is no one universal answer to these questions, so you'll basically need to pick one schema, try it, and see if it works and feels fine for your purposes in case of pretty bad connections.

---

[2] that is, beyond capabilities of input buffer [[TODO!: add input buffer to Fig VII.2/VII.3]]
[3] and if keyboard state change has happened, it can and SHOULD be combined with the acknowledgment IP packet to save on bandwidth, but this is a bit different story, discussed in Chapter [[TODO]]

Input Timestamping

One issue which is often associated with inputs, is client-side input timestamp (in practice, usually it will be a tick-stamp). This is indeed necessary to facilitate things such as Lag Compensation described in "Lag Compensation" subsection above. On the other hand, as soon as server starts to trust this timestamp, this trust (just as about any kind of trust out there) can be abused. For example, if within your game you have a Good-Bad-Ugly-style shootout, and compensate for the lag, then the Bad guy, while having worse reaction, could compensate for it by sending "shoot" input packet with an input timestamp which is 50ms earlier than the real time, essentially gaining an unfair advantage for these 50ms. In general, such cheating (regardless of way of implementing it[4]) is a fundamental problem of *any* kind of lag compensation, so you should be really sure how to handle various abuse scenarios before you introduce it.

"For example, you have a Good-Bad-Ugly-style shootout, and compensate for the lag, then the Bad guy, while having worse reaction, could compensate for it by sending "shoot" input packet with an input timestamp which is 50ms earlier than the

---

[4] no, measuring pings instead of relying on input timestamps doesn't prevent the cheat, it just makes the cheat a bit more complicated

"Macroscopic" Client Actions

In addition to sending bare input to server, client usually needs to implement some actions which go beyond it.

Examples of such “macroscopic” actions include such sequences of inputs as:

real time, essentially gaining an unfair advantage for these 50ms.

- player looking at object (usually processed purely on client-side)
- client showing HUD saying that “Open” operation is available because object under the cursor is container (again, processed purely on the client-side)
- client pressing “Action” button (which means “Open” in this context)
- client showing container inventory (obtained via an RPC call, or taken from Publishable State)
- player choosing what to take out
- only then client invoking a Client-to-Server RPC such as take_from_container(item_id, container_id)

For such RPC calls as take_from_container(), disconnect during the call can be simply ignored in most cases (so that player will need to press a button again when/if the connection is restored)

Another set of “macroscopic” actions (usually having even longer chains of events before RPC call is issued) is related to dialog-based client-side interactions such as in-game purchases. In these cases, all the interactions (except, maybe, for some requests for information from the server) usually stay on the client-side until the player decides to proceed with the purchase; when this happens, Client-to-Server RPC call containing all the information necessary to proceed with the purchase, is issued.

For such RPC calls, handling of disconnect during an RPC call is not that obvious. If you want to be player-friendly (and usually you should be), you need to consider two scenarios. The first one is when the disconnect is transient, and client is able to reconnect soon; then, you need a mechanism to detect whether your RPC call has reached the server, to get the result if it did, and to re-issue the call if it didn’t; this would allow to make disconnect look really transient for the player, and to show the result of the purchase as if the disconnect has never occurred. To implement it, you’ll need to implement both re-sending of RPC call on the client side, and dealing with duplicates on the server-side, in a manner similar to the one described in “Server-to-Server” Communications section below.

The second scenario occurs when the RPC call is interrupted by disconnect before obtaining the reply, and disconnect takes that long that client gets closed (or server gets restarted). In this case, the only things we can practically do for the player, are not directly related to the communication protocols (but they still need to be done). Two most common features that help to make player not *that* unhappy in this second scenario, are (a) to send her an e-mail if the "purchase" RPC call has reached the server (it doesn't help to vent frustration if the call didn't reach the server ☹), and (b) to provide her with a way to see the list of all her purchases from the client when she's back online (which we need to do anyway if we want to be player-friendly).

"The second scenario occurs when the RPC call is interrupted by disconnect before obtaining the reply, and disconnect takes that long that client gets closed (or server gets restarted).

Server-to-Client

While server does send a lot of information to client (both as a part of Publishable State, and as replies to Client-to-Server RPC calls), it is not too common to call RPC from the server side.[[TODO!: add note to Chapter VI/"Asynchronous" that it is not too common to do it this way, and that it is usually client-side-driven rather than server-side-driven]]

On the other hand, in some cases such RPC calls (especially void RPC calls without the need to process reply on the server side) are helpful. One such example is passing pocket cards to the client in a poker game. This will allow to exclude pocket cards from Publishable State (which in absence of Interest Management allows for rampant cheating, as was described in "Interest Management: Traffic Optimization AND Preventing Cheating" section above).

Server-to-Server Communications

As noted above, from the point of application layer Server-to-Server communications can be made seamless (hiding disconnects, including those resulting from FSM relocations, from application layer). However, this comes at the cost of infrastructure level doing this work behind the scene. One fairly common protocol which does achieve seamless handling of disconnects, implements two related but distinct features.

First, as noted above, we'll be usually dealing with "non-blocking RPC calls" anyway. To support some kind of callback (whether being OO, lambda, or future), we'll need to keep a list of "outstanding RPC requests" (with their respective IDs) on the caller side anyway. And as soon as we have this list of "outstanding RPC calls", we have sufficient information to re-send RPC request in case of lost packet/disconnect.[5]

Idempotence

Idempotence is the property of certain operations in mathematics and computer science, that can be applied multiple times without changing the result beyond the initial application.
— Wikipedia —

On the other hand, this technique, while guaranteeing that we will get *at least one* RPC request on the callee side for each RPC call on the caller side, doesn't guarantee that it will be *the only one*. In other words, if implementing only the logic described above, duplicate RPC calls on callee side can happen for a single RPC call on the caller side. While making all the RPC calls *idempotent* would solve this problem, in practice making sure that each and every call is idempotent at the application layer, is not exactly realistic.

That's why a second part of processing (this time – on the callee side) needs to be added. For example, we can make the callee side keep the list of "recently-processed RPC request_ids" (with associated replies), and if some request with an ID from this list comes in – we should just to provide the associated reply without calling anything on the callee side. This scenario may legitimately happen if the connection was lost-and-restored after the request was received, but before the reply was acknowledged, but the handling mentioned above, guarantees that everything is handled "as if" disconnect has never happened.

As soon as we have these two parts of processing (in practice, it will be a bit more complicated, as information on "which replies can be dropped from the list" will need to be communicated too, plus, most likely, we'll need to implement handshakes to distinguish between new connection and the broken one) – we can say that our Server-to-Server communication is tolerant to all kinds of transient inter-server disconnects. This is necessary not only to deal with inter-server disconnects at TCP level (which are extremely rare in practice), but is also one of prerequisites to deal with scenarios when we're restoring/moving an FSM (see Chapter VI, section "Failure Modes and Effects" for details).

"As soon as we have these two parts of processing – we can say that our Server-to-Server communication is tolerant to all kinds of transient inter-server disconnects.

An alternative (similar, but not identical) way of dealing with such transient-disconnect issues, is to create two "guaranteed delivery" message streams (going into opposite directions), with each of the streams keeping its own list of "unacknowledged messages", and re-sending them on loss-and-restore of underlying connection; on the receiving side, a simple "last ID processed" is sufficient[6] to filter out all the duplicates.

[5] as noted in Chapter VI, section “On Inter-Server Communications”, we’ll probably use TCP for inter-server communications anyway, so such re-send will need to happen only on TCP disconnect/reconnect
[6] that is, assuming that message IDs are guaranteed to be monotonous

[[To Be Continued…

This concludes beta Chapter VII(c) from the upcoming book “Development and Deployment of Massively Multiplayer Games (from social games to MMOFPS, with social games in between)”. Stay tuned for beta Chapter VII(d), “IDL: Encodings, Mappings, and Backward Compatibility”]]

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *MMOG: World States and Reducing Traffic*

Filed Under: Distributed Systems, Network Programming, Programming, System Architecture
Tagged With: client, game, multi-player, network, protocol, RPC, server

Copyright © 2014-2016 ITHare.com