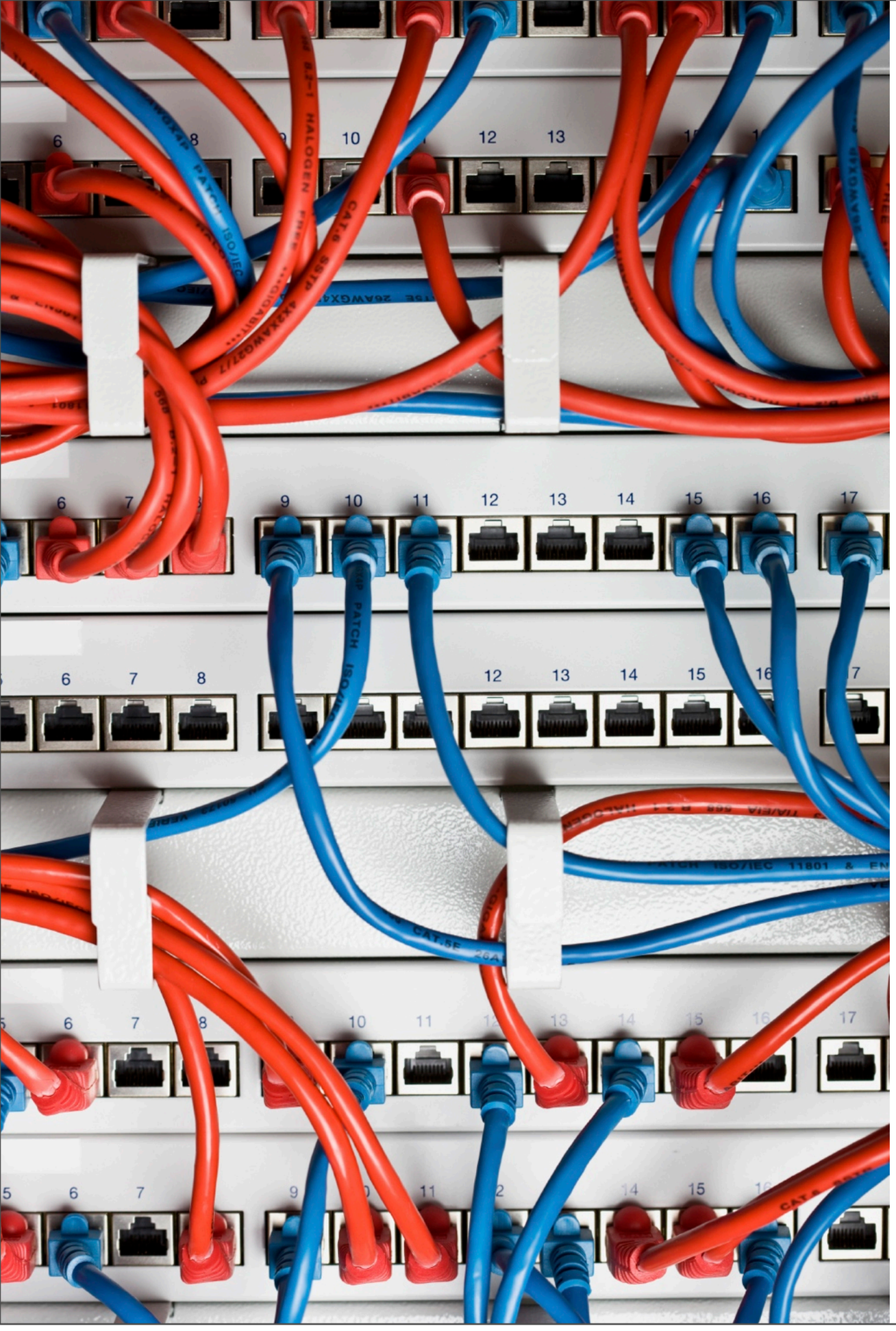


Robust Efficient Networking

Ben Garney
Pushbutton Labs
beng@pushbuttonlabs.com

- My name is Ben Garney, and I am a programmer.
- Developer at GG for 5 years, now at PBL
- Worked on TGE, TGEA, future tech stuff, and TNL
- This talk may sound familiar... (Frohnmayer00)

Dude Hopper V

Connect

Options

Exit

Anyway, let’s start with an example. We start our favorite game...

Dude Hopper V

Connect

Options

Exit

Anyway, let’s start with an example. We start our favorite game...

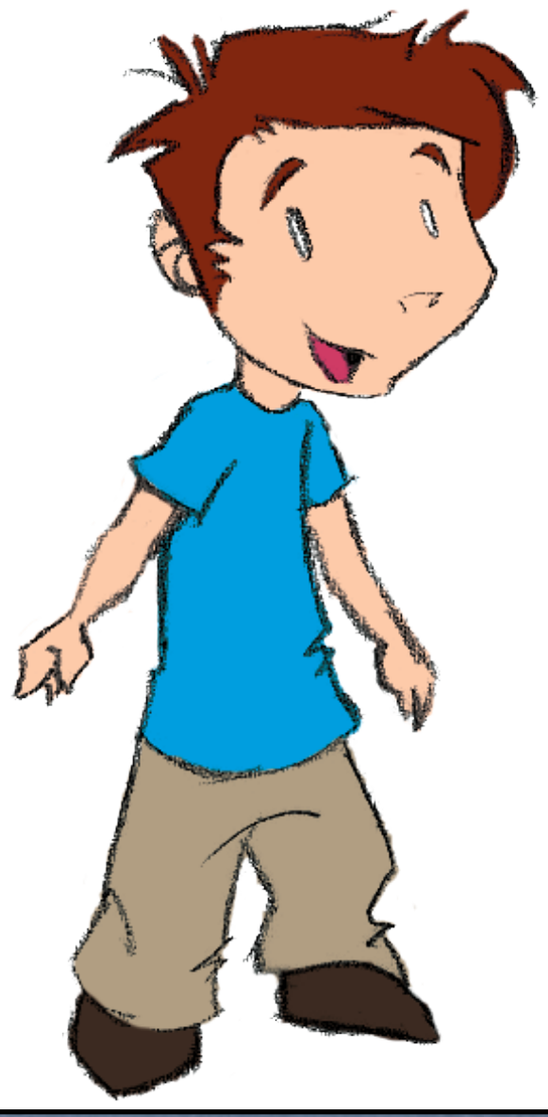
Dude Hopper V

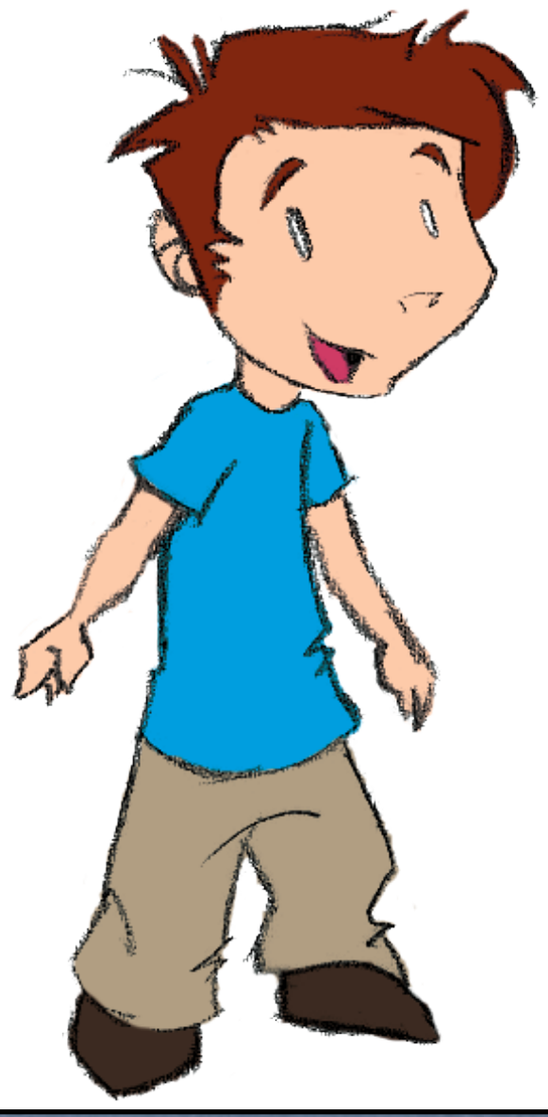
Connect

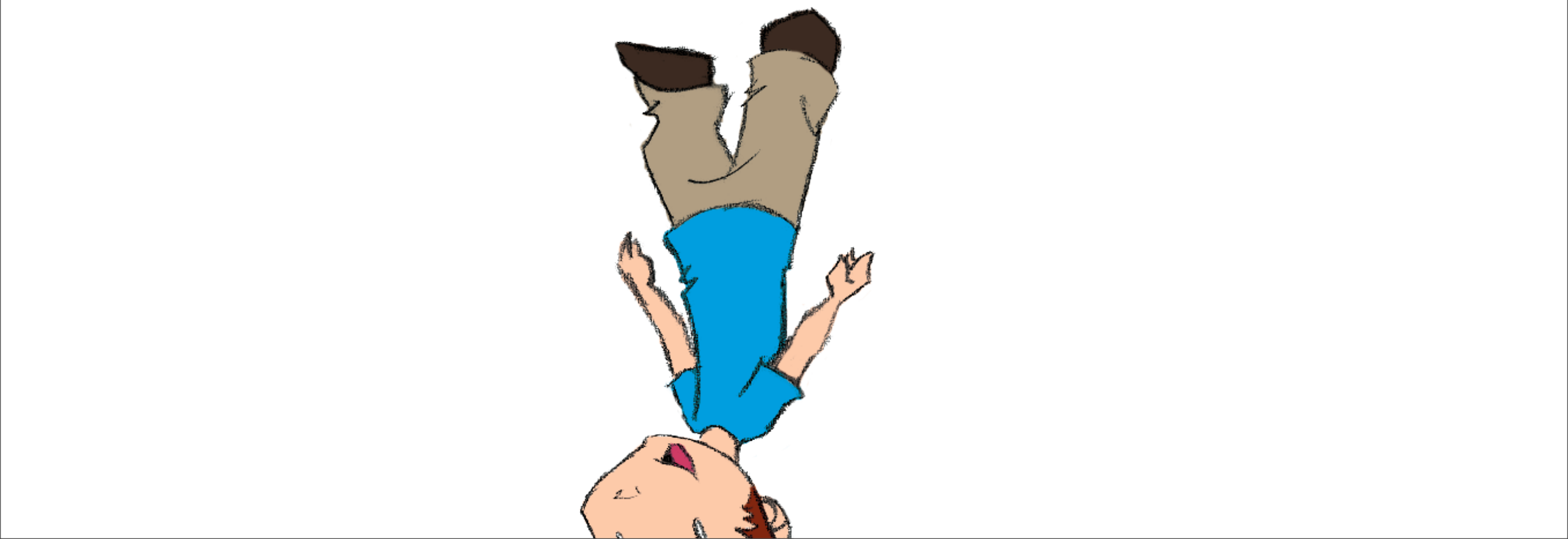
Options

Exit

Anyway, let's start with an example. We start our favorite game...







GAME OVER

This talk is about how to avoid that.

For action game developers... Should be obvious by now, you need good networking! State of the art is high!

For MMO folks... You are paying for every byte of bandwidth. You are limited by players/server.

Our Goals

Robust

Efficient

Responsive

Robust, it has to deal with packet loss, cheaters, data corruption, and so forth.

Very efficient encodings, every extra few bytes is another update we can send. Even though everyone has cable modems now, other forces keep things lean (loss, latency, server-side bandwidth)

Responsive, players must feel that they are in control at all times or immersion is lost.

www.opentnl.org

The architecture I'm going to describe is based largely on the Torque Networking Library. It solves all these problems well. It's a free, GPL library, so feel free to look at it and learn on your own.

Conceptually, OpenTNL is the third generation of this architecture, the first being Tribes, and the second being Tribes 2.

This is the server. The server sends a packet out over the wire.

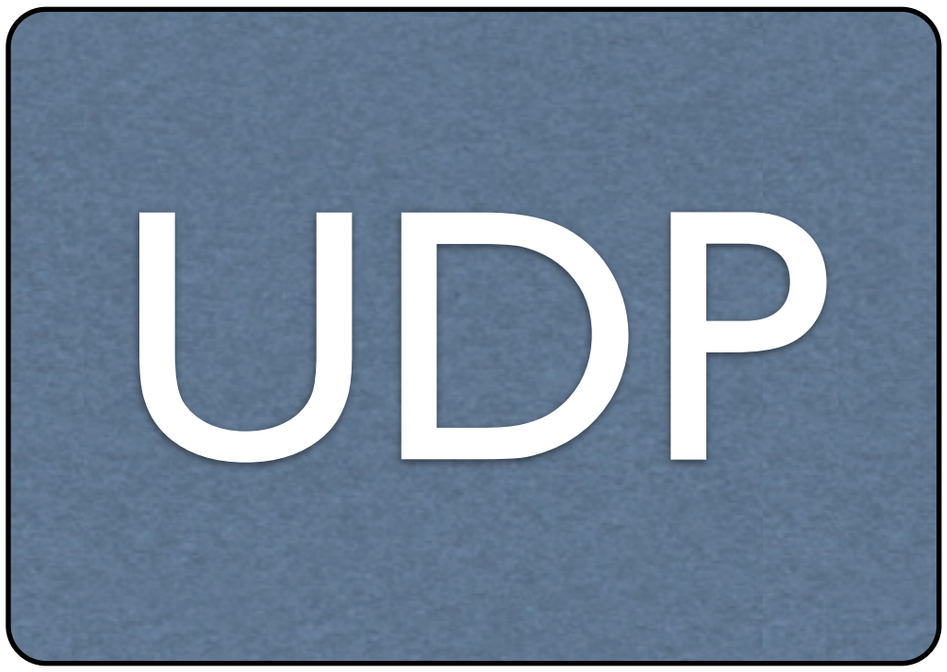
Protocol Layers
Bit Stream
Packet Notify
Events
Ghosts
Control Object
[Your Data Here]

What does it look like?

Packet Notify

The Packet Notify protocol gives us the power to act intelligently in the face of adverse network conditions.

- UDP gives us unguaranteed delivery.
- TCP gives us guaranteed delivery.
- Need middle ground - guaranteed to know if packet has been delivered yet.

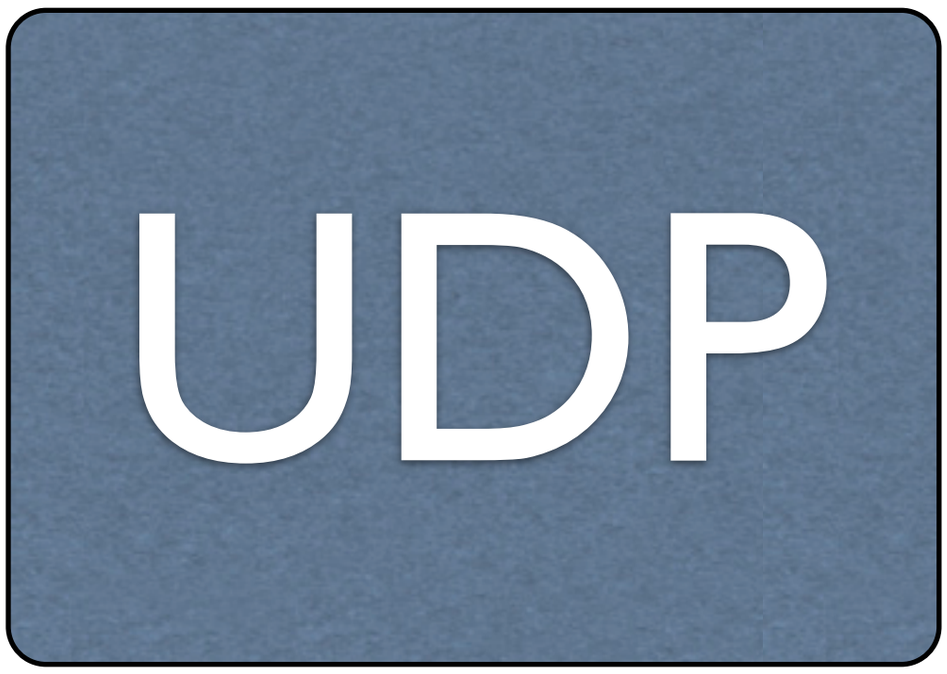


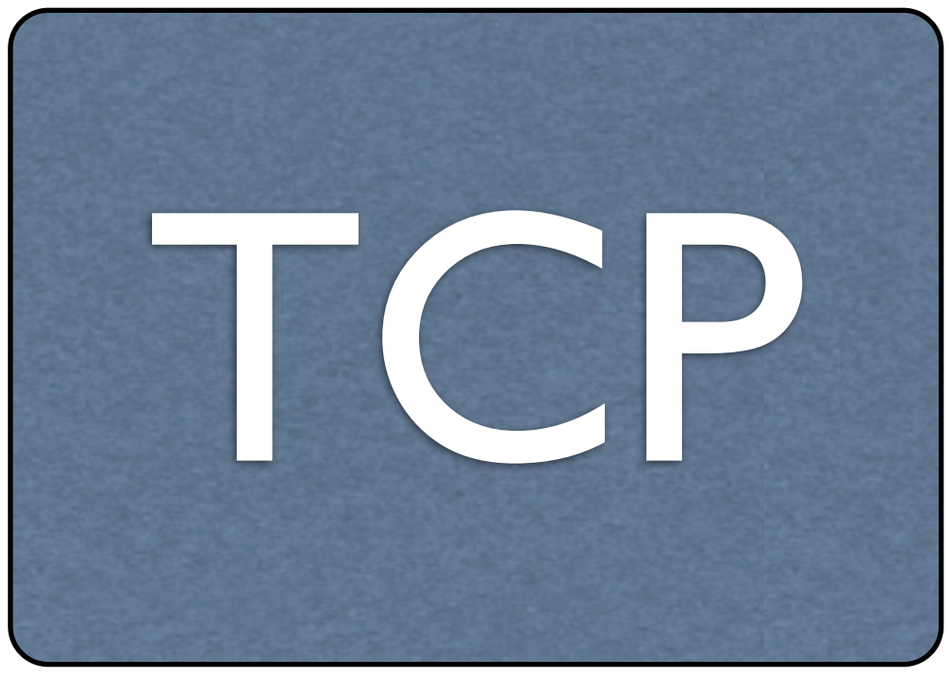
- UDP gives us unguaranteed delivery.
- TCP gives us guaranteed delivery.
- Need middle ground - guaranteed to know if packet has been delivered yet.

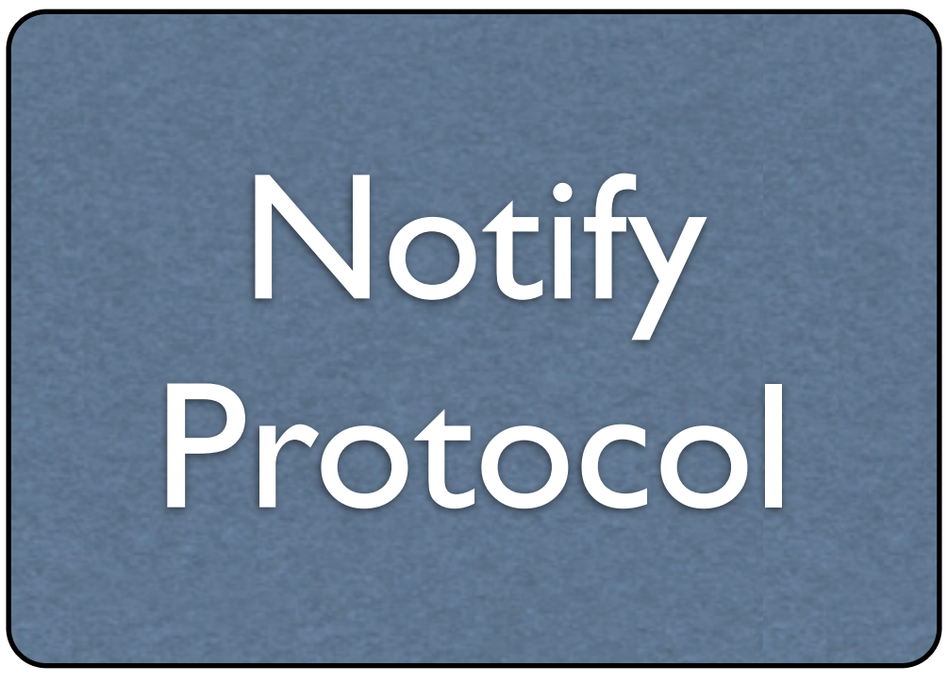
UDP

TCP

- UDP gives us unguaranteed delivery.
- TCP gives us guaranteed delivery.
- Need middle ground - guaranteed to know if packet has been delivered yet.



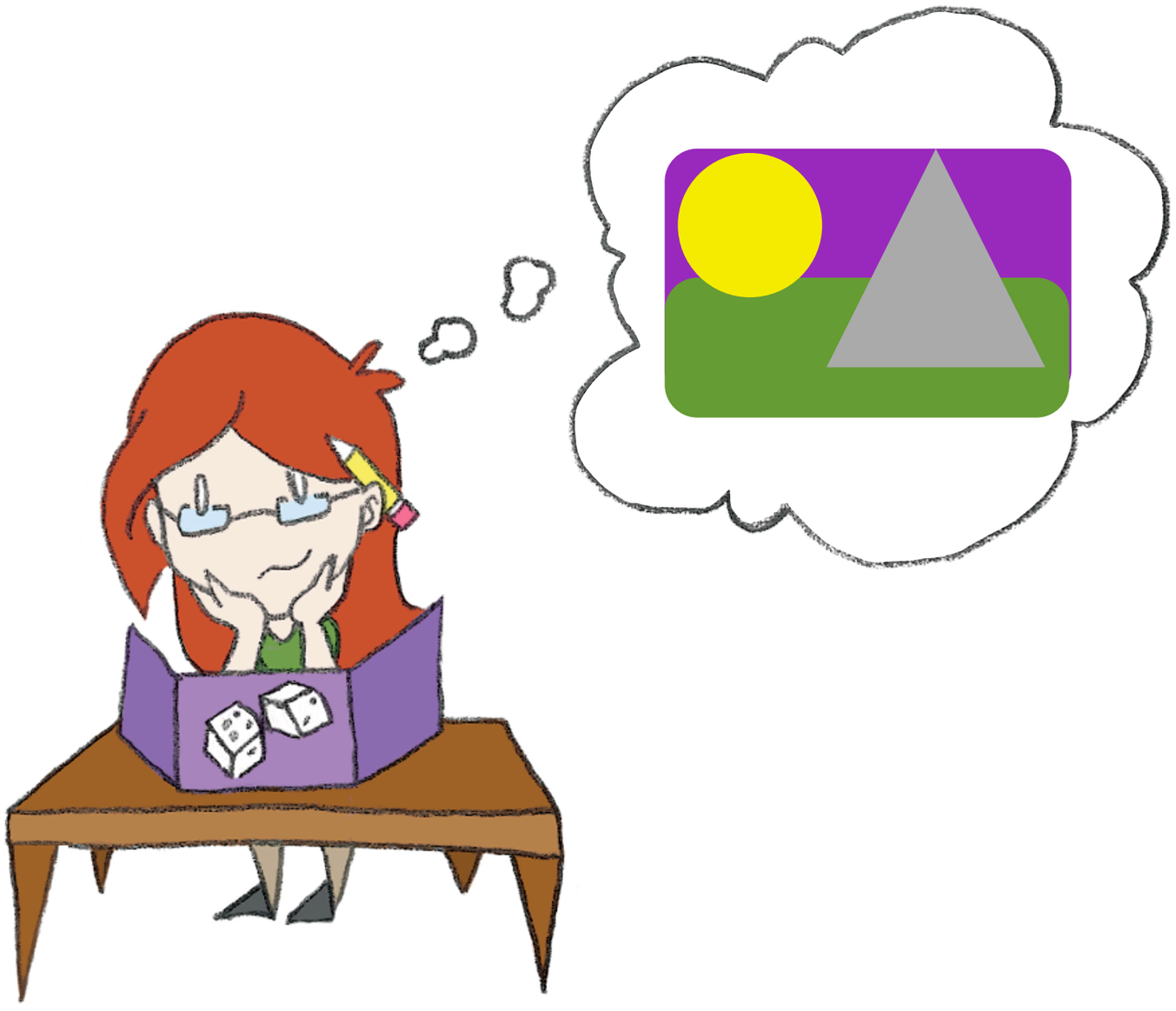


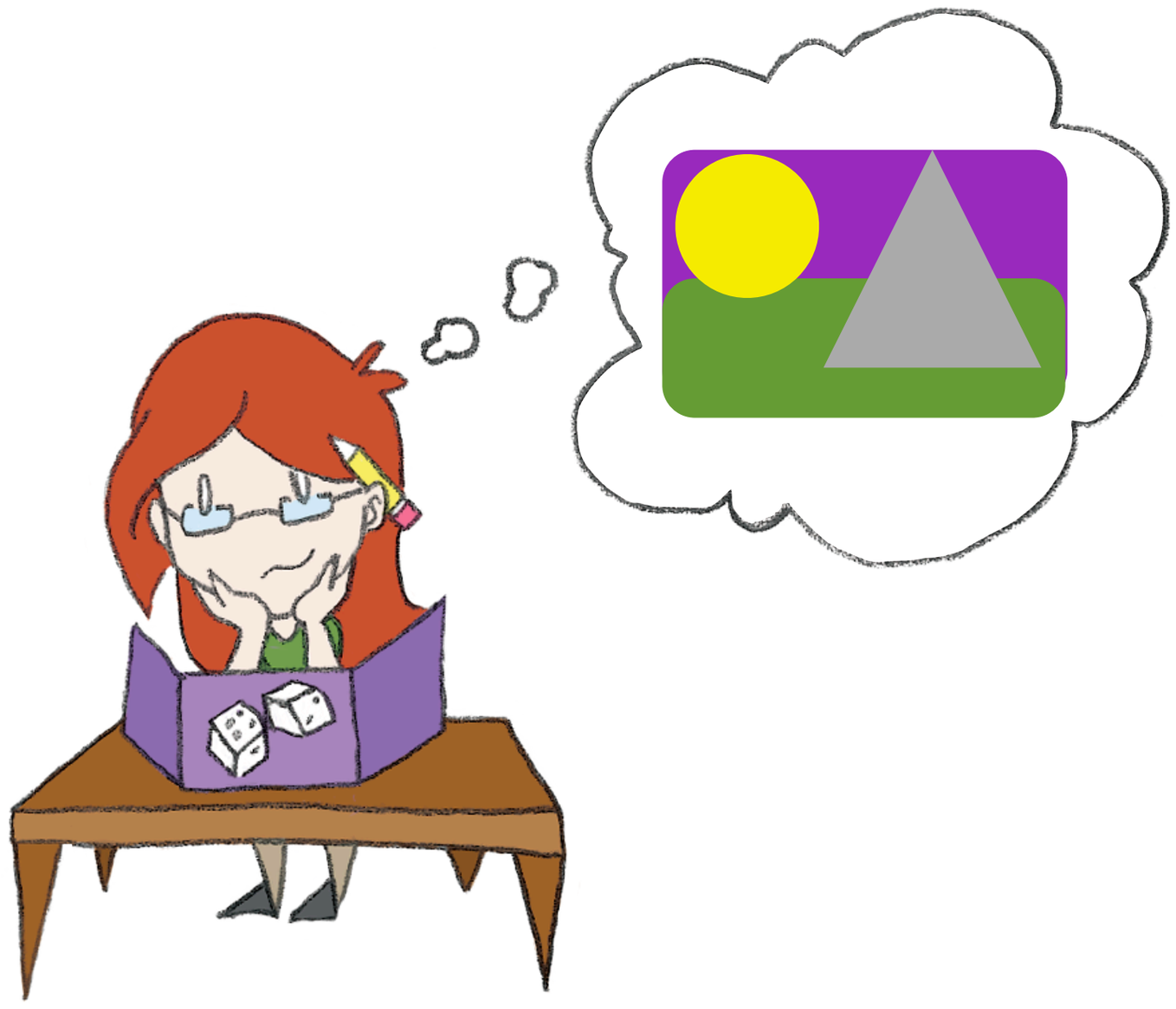


- UDP gives us unguaranteed delivery.
- TCP gives us guaranteed delivery.
- Need middle ground - guaranteed to know if packet has been delivered yet.

Demo: Changing the color of the sky.

How do we deal with resending? UDP vs. TCP vs. Notify





Let’s go back to when the server was sending that message...

What if a router en route to the client loses a packet?

Each protocol is going to handle this differently...

UDP does nothing!

TCP lets the OS deal with it. So the data shows up eventually... but the OS won't do you any favors by telling you about it.

Notify is implemented over UDP, but ...

Packet ID	34
Last Received Packet ID	495
Notify Window	10110 1111 0111 0101
Payload	1010 0101 1111 0000 0110 1000 1101 0100 1010 0101 0011 0001 0000 1100 0110 1001 ... etc ...

This is roughly the data stored in each packet. The key thing is the notify window. Each bit corresponds to whether the peer sending you the packet has received that packet yet. To clarify...

Hi Server!

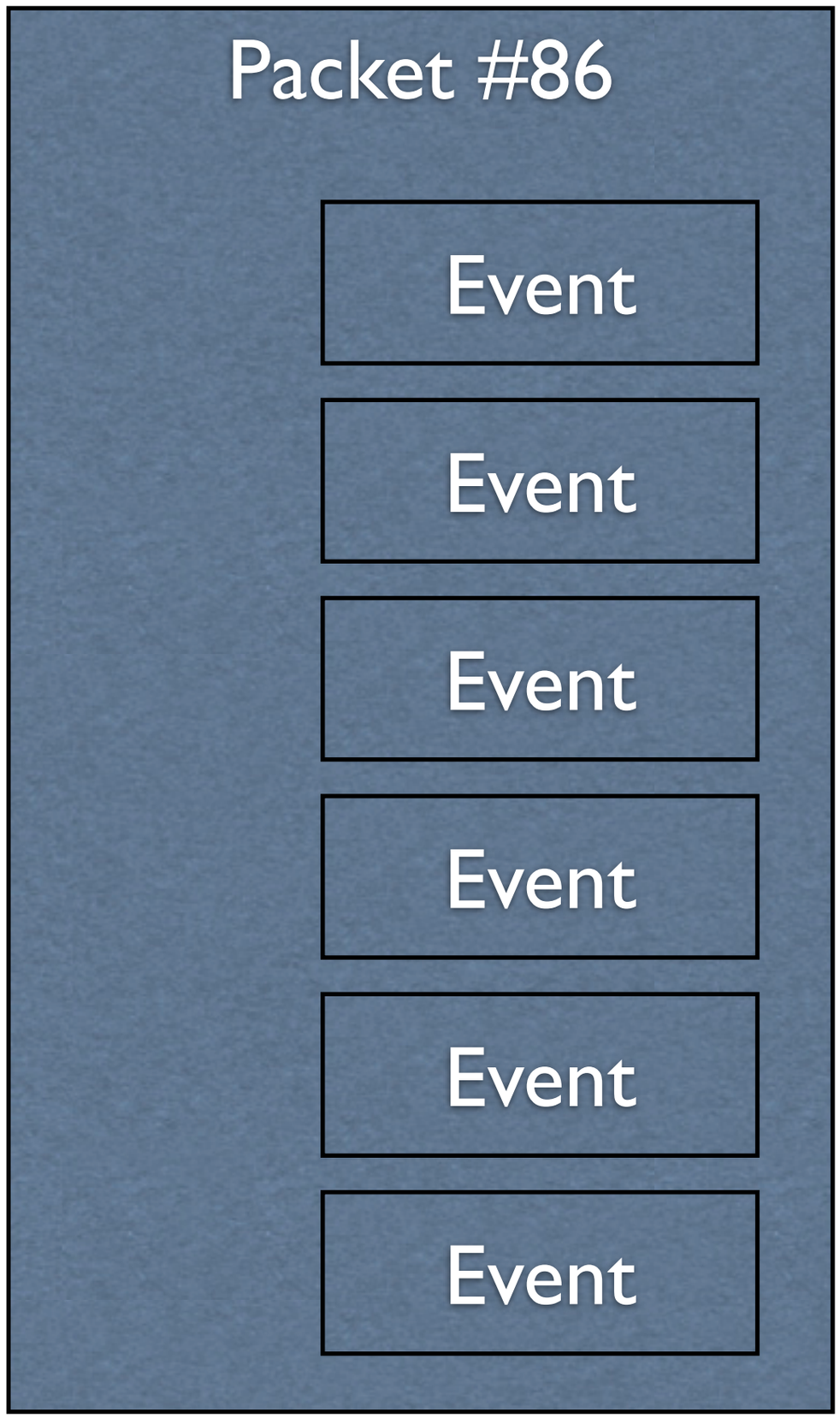
This is message #49 from me (the client).

The last message I saw from you was #91.

I saw message #90.
I didn't see message #89.
I saw message #88.
I saw message #87.
I didn't see message #86.
I saw message #85.

Your Pal,
Client

The messages read something like this. But it's sent in a few bytes instead of all this space.



Whenever they send a packet, they keep references to what was in it. In this case, since the client didn't see packet #86, the server can notify each event that it was not received.

This is the key difference between TCP/UDP and the notify protocol. You can be intelligent when bad stuff happens.

Events

Server-side

- Pack data
- Get notified if it made it.
- Optionally resend.

Events are pretty simple, but there are a few key pieces.

First, once you've sent it off, you'll guaranteed find out if it made it or not. In TNL, the protocol can automatically enforce some rules (like guaranteed delivery, or guaranteed ordered delivery).

Client-side

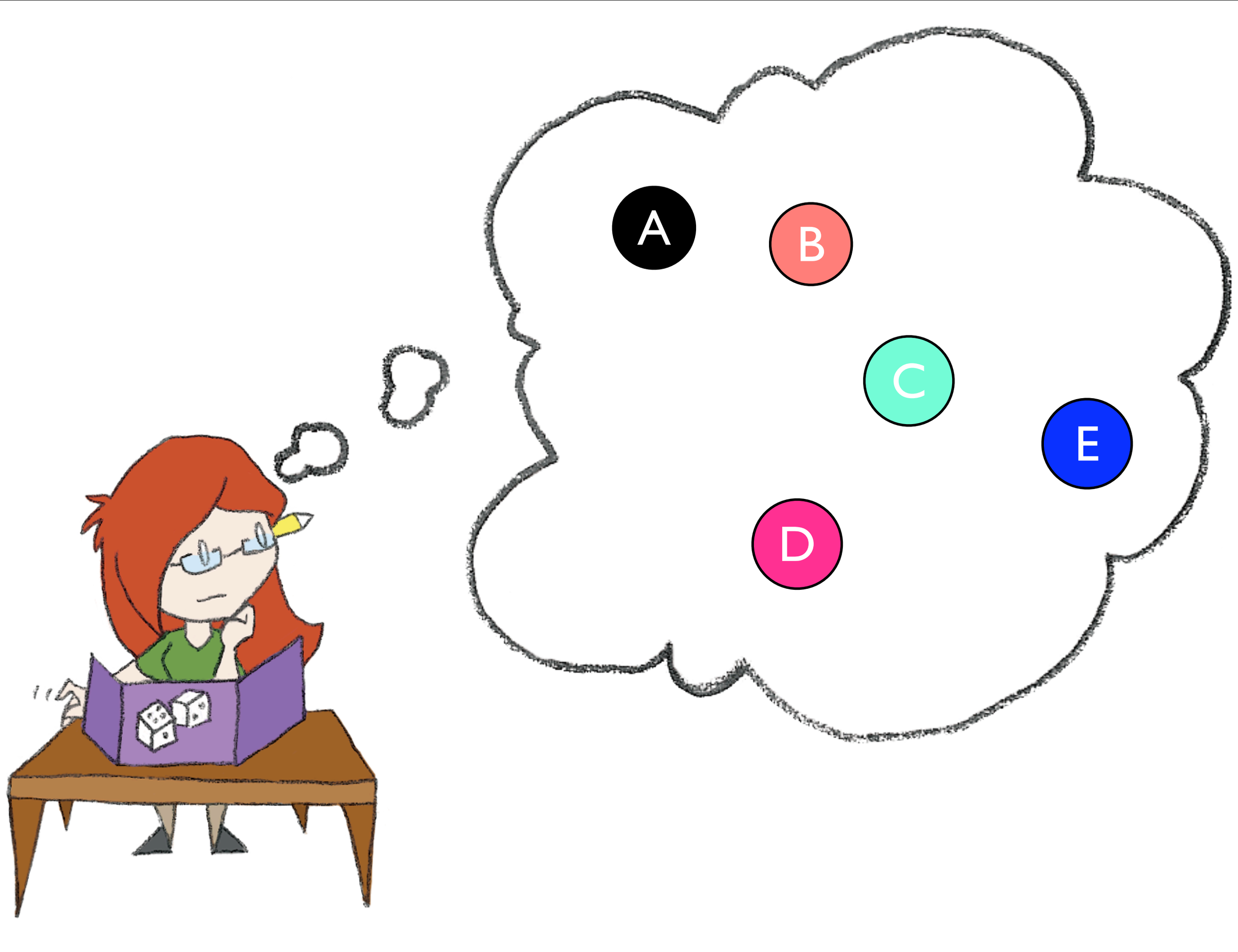
- Unpack data.
- Run handler.

On the client side, it’s even simpler, you get the data and run a handler. You can easily make this into an RPC system, use it for chat, important game events, etc.

Ghosts

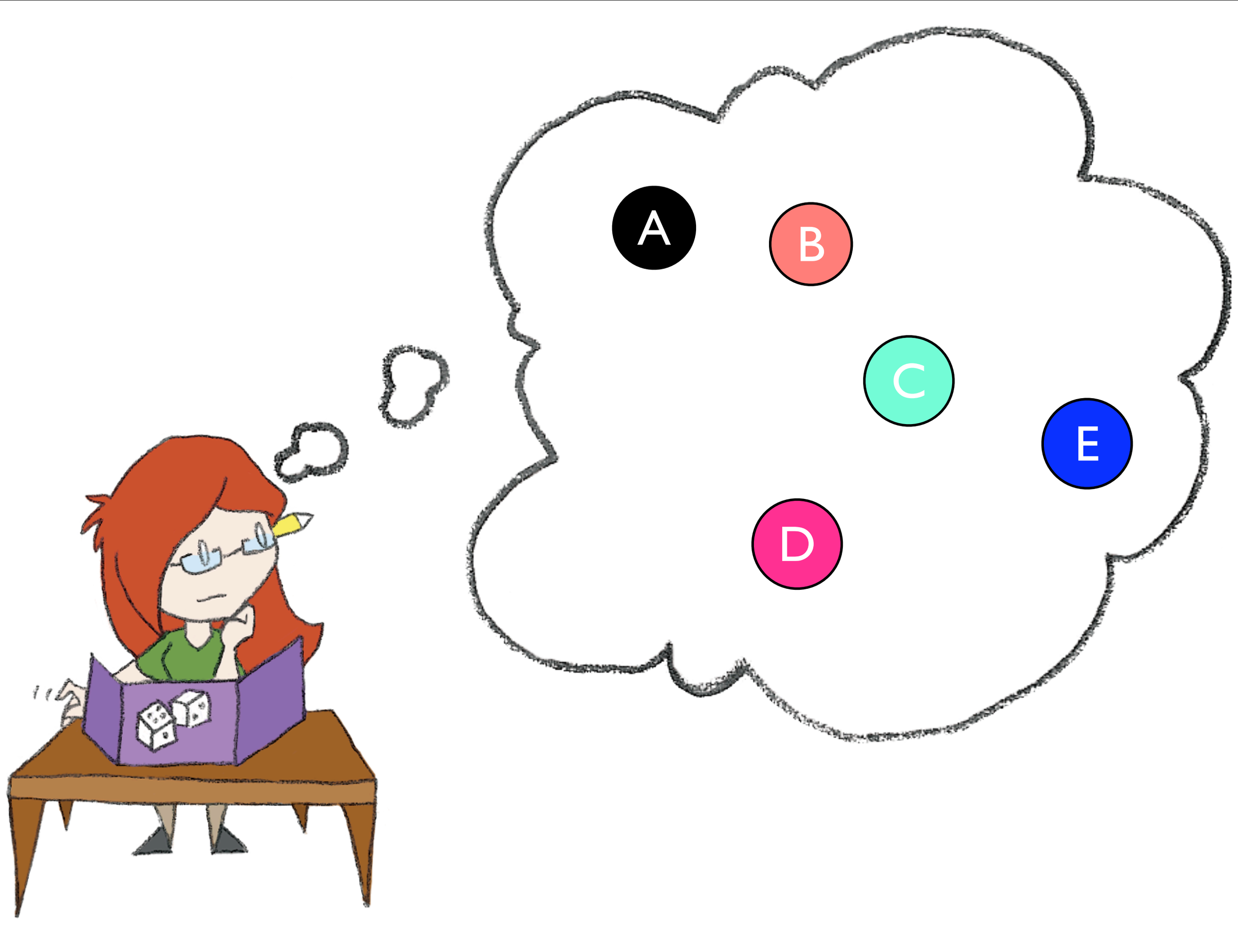
AKA Most Recent, Proxies, etc.

Synchronize state of objects in one simulation to another simulation. The most recent state is always sent, so you always end up in the right place, but may not follow the same path. (More on that later.)



Let’s explain the system with an example. For the purposes of our discussion we’ll talk about client and server, but ghosting is just a one-way communication. You can do it both ways (on different objects).

So we have the server, which is maintaining some game simulation with objects (A, B, C, D, E) in it.

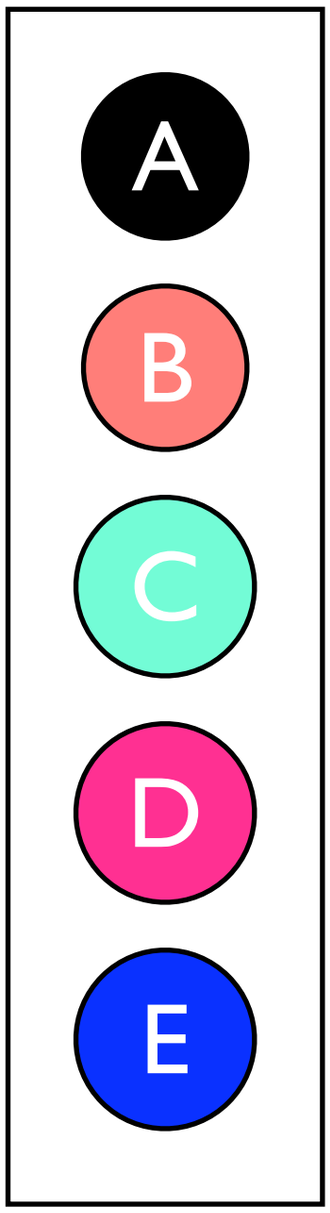


Now, what we'll do is send a packet to the client at fixed intervals – say ten times a second – containing bit-packed state information for each object.

Object A
Position
Heading
Color

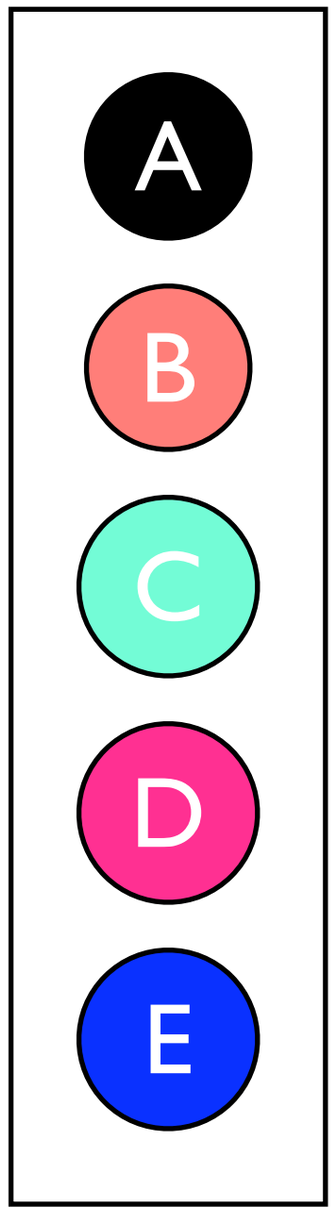
An individual update looks something like this...

What we do is track when state changes, and only send “dirty” data. Each of these boxes might consist of several different values. Usually we group states that will be dirty at the same time together.



And once it’s all in a packet, it looks like this.

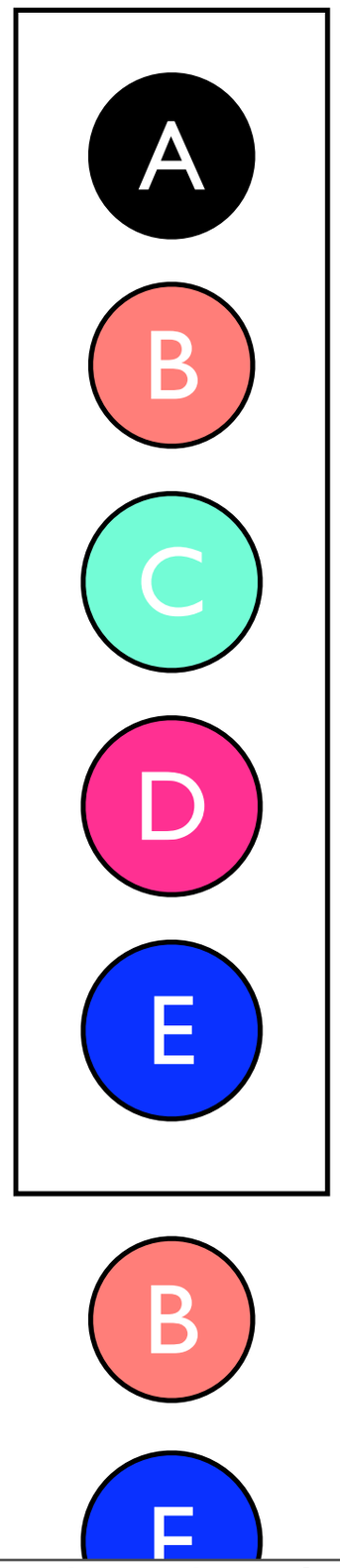
There are some problems here.



One problem is what happens when a hacker (either the client or someone intercepting traffic) gets ahold of the packet? They know the whole world state. This makes cheating easy.

The second problem is - what do you do once the packet is full and you have more things to update?

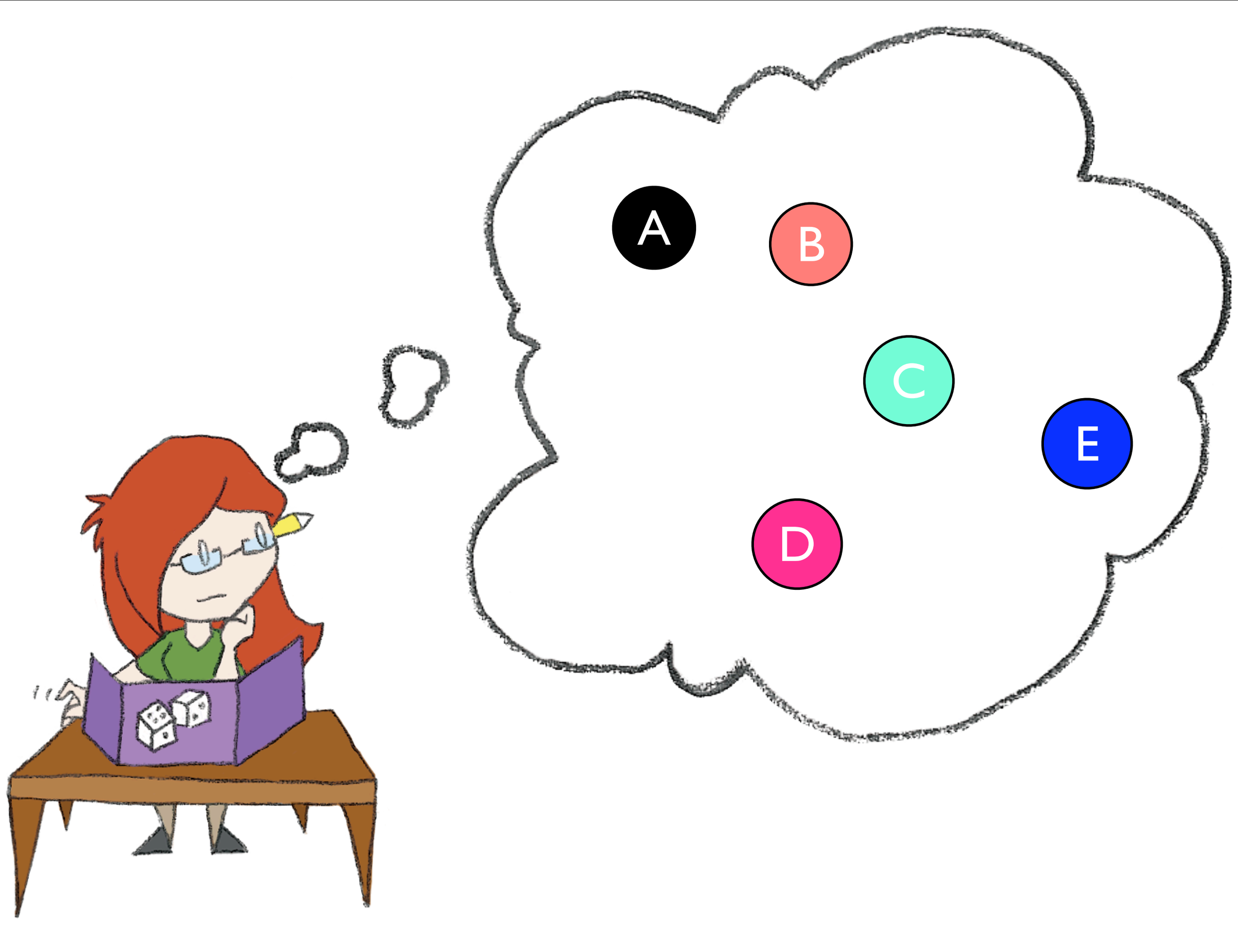
With Blockland's 85k bricks, this is an obvious issue. But your MMO - with dozens or hundreds of dynamically placed objects - will run into it, too. In fact, all but the simplest games will have problems with this.



One problem is what happens when a hacker (either the client or someone intercepting traffic) gets ahold of the packet? They know the whole world state. This makes cheating easy.

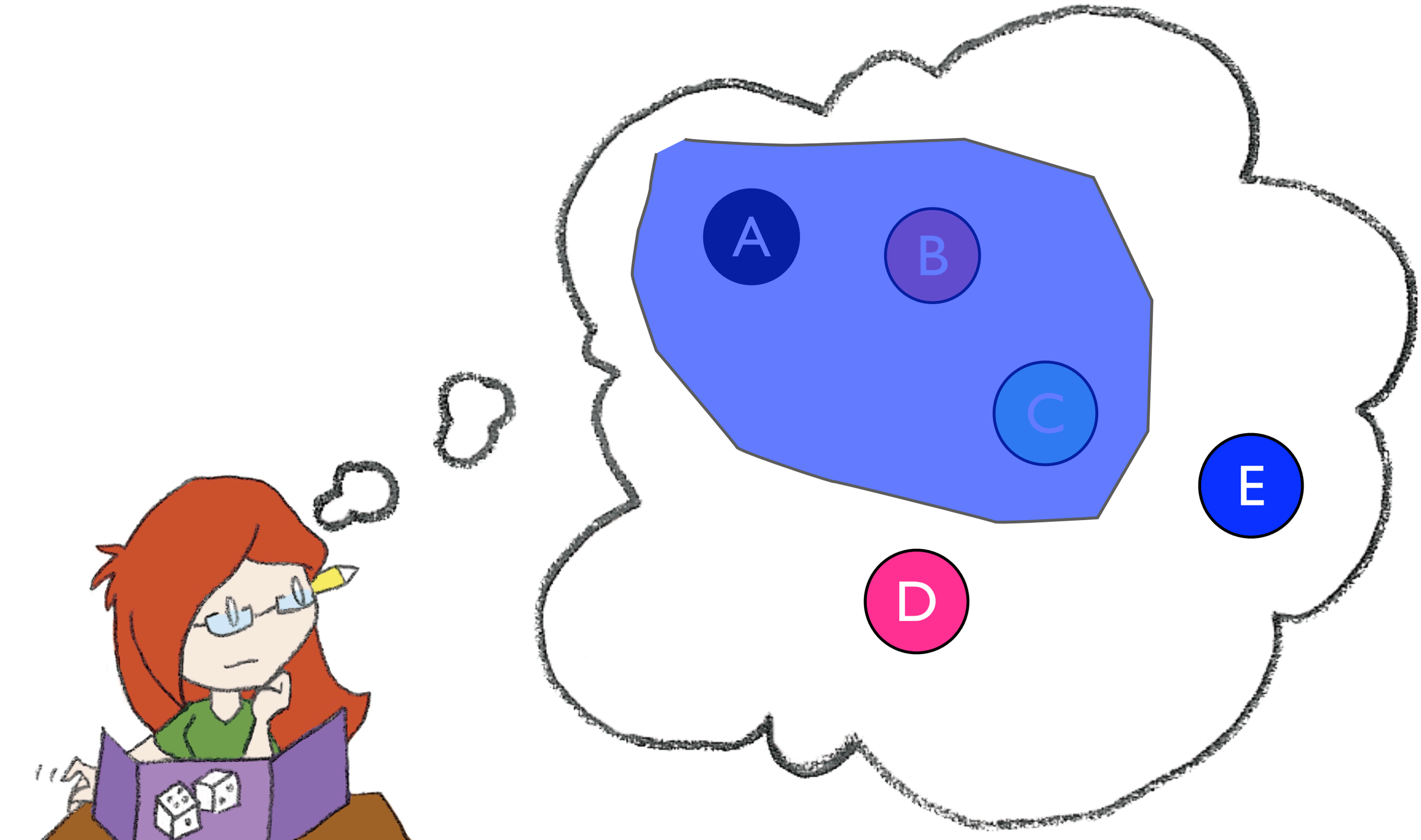
The second problem is - what do you do once the packet is full and you have more things to update?

With Blockland's 85k bricks, this is an obvious issue. But your MMO - with dozens or hundreds of dynamically placed objects - will run into it, too. In fact, all but the simplest games will have problems with this.



We determine if objects are in scope. Only scoped objects are transmitted.

Additionally, at this time, we calculate priority. The highest priority objects in scope are sent first. Distance is a good metric. We also include time since the last update, so that eventually everything gets updated. Now our packet looks like this...

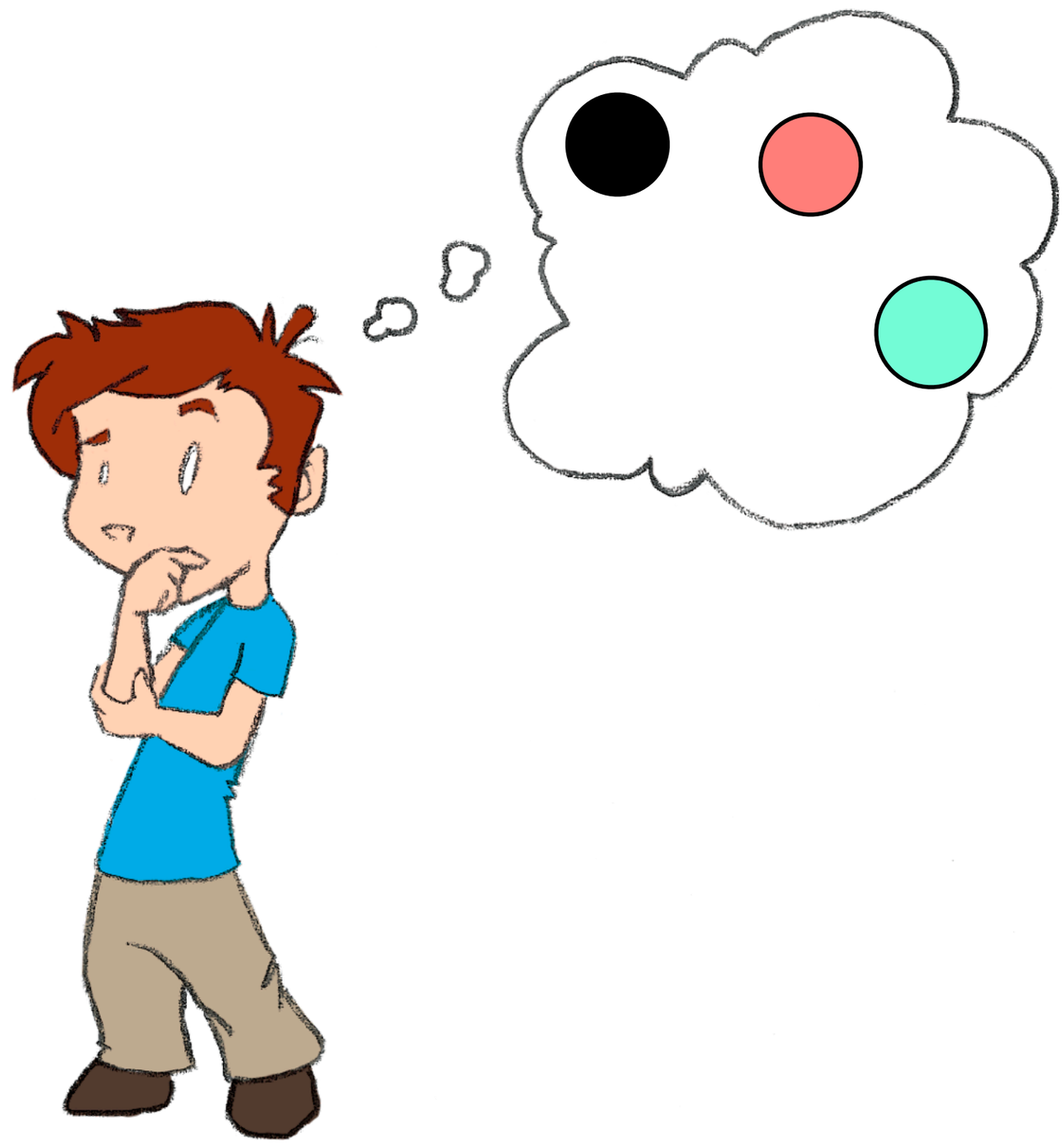


We determine if objects are in scope. Only scoped objects are transmitted.

Additionally, at this time, we calculate priority. The highest priority objects in scope are sent first. Distance is a good metric. We also include time since the last update, so that eventually everything gets updated. Now our packet looks like this...

Plenty of room. Of course, if we had a hundred objects in the world, the packet would be full all the time, and prioritization would be hugely important. Regardless, we've saved time and bandwidth.

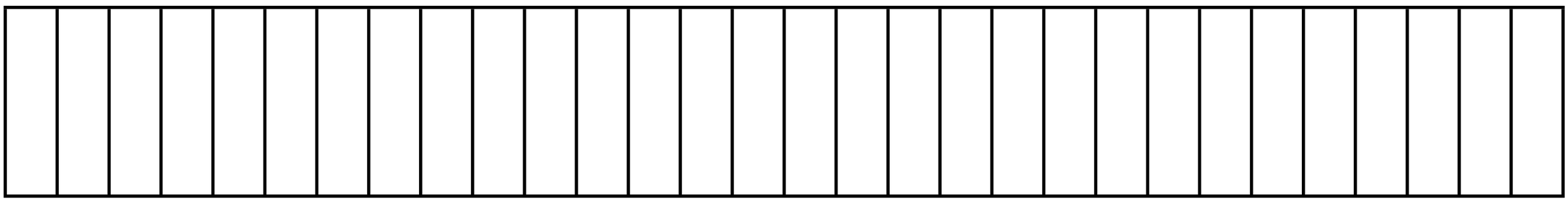
In addition, now cheaters can't get at information they shouldn't, because it never leaves the server.



When the client receives the update, then he can reconstruct the world state.

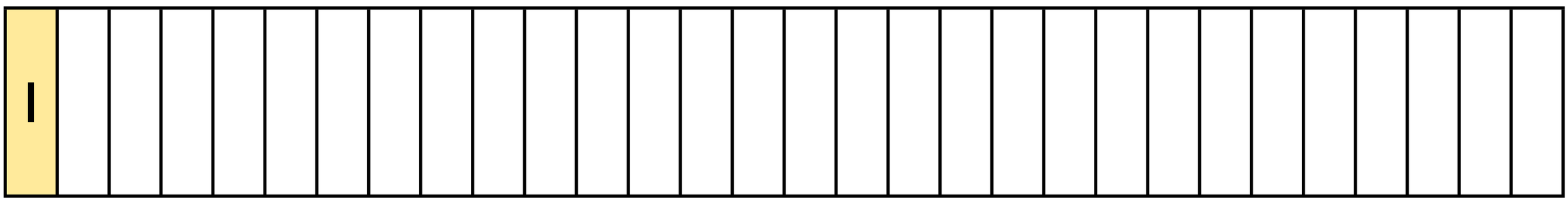
Bit Stream

Why are bit streams important? Bit packing lets you control how much range/precision your data uses. They are also very efficient to produce.



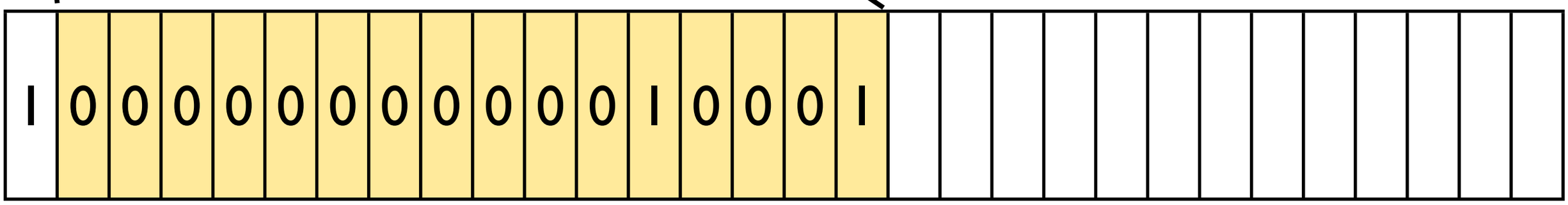
So here’s a bit buffer

writeFlag(1);



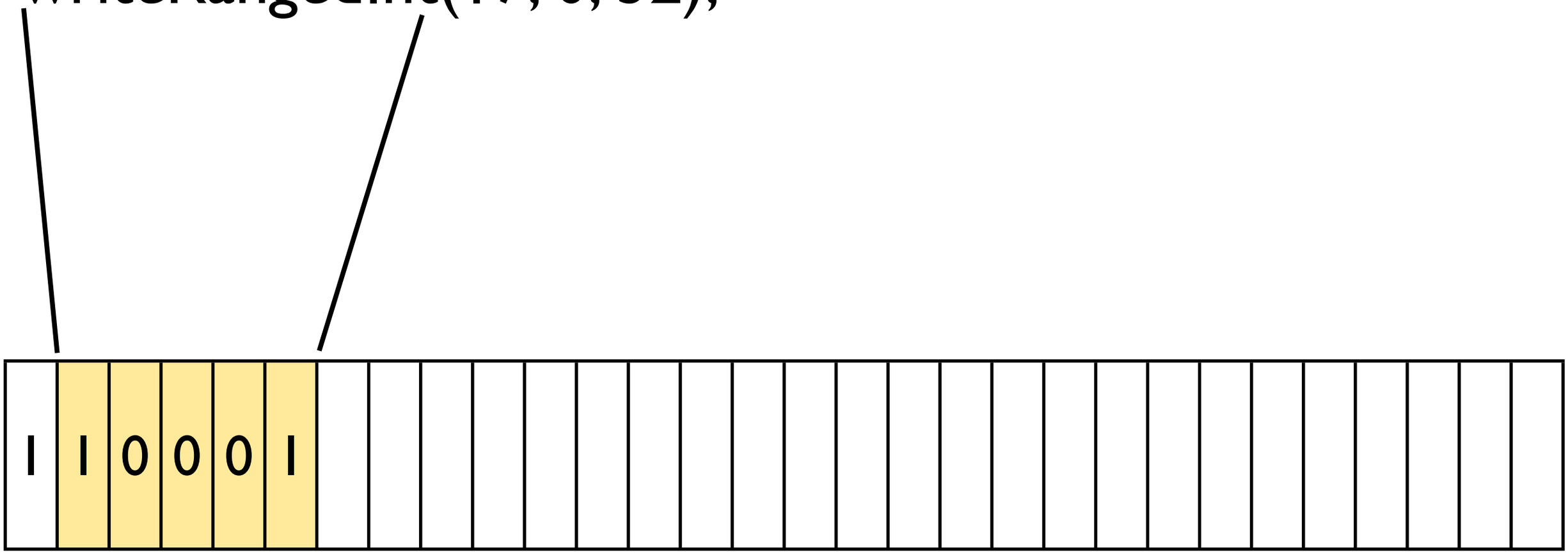
and we can store a flag in it

```
writeFlag(1);
writeInt(17);
```



or we could store an integer... this is a 16 bit integer

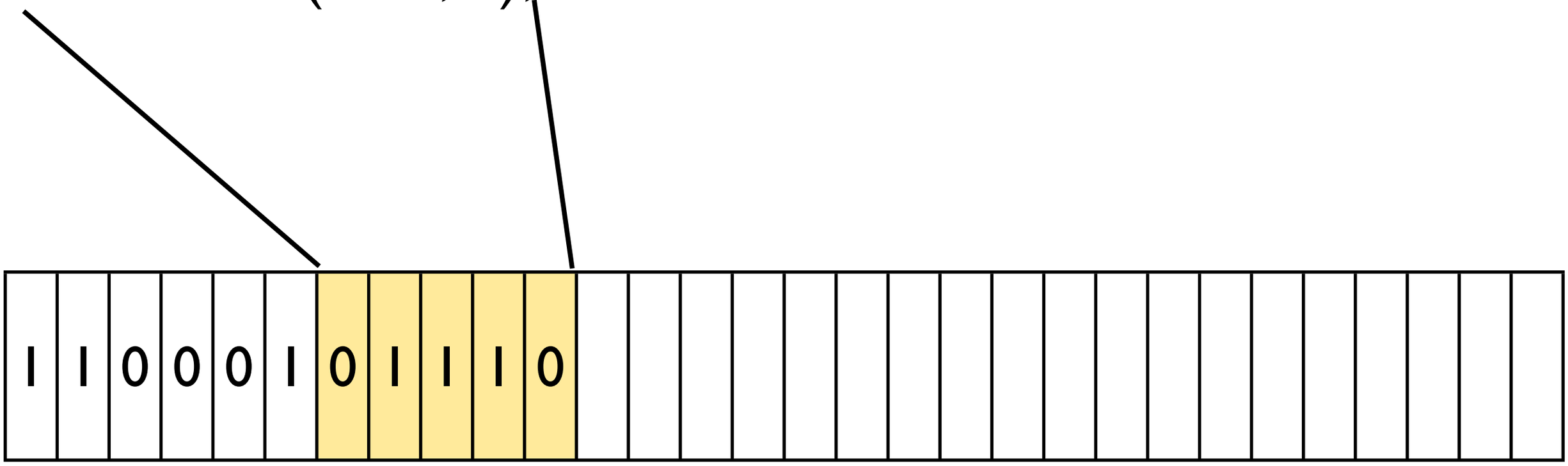
```
writeFlag(1);
writeRangedInt(17, 0, 32);
```



the nice thing about bit level control is we can store just a 5 bit integer if we know the range.

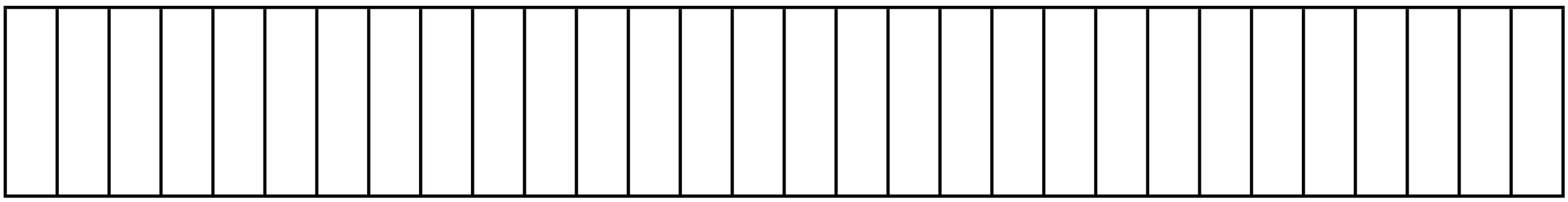
(you can do better than this with arithmetic coders, but they’re trickier to work with)

```
writeFlag(1);
writeRangedInt(17, 0, 32);
writeFloat(0.45, 5);
```



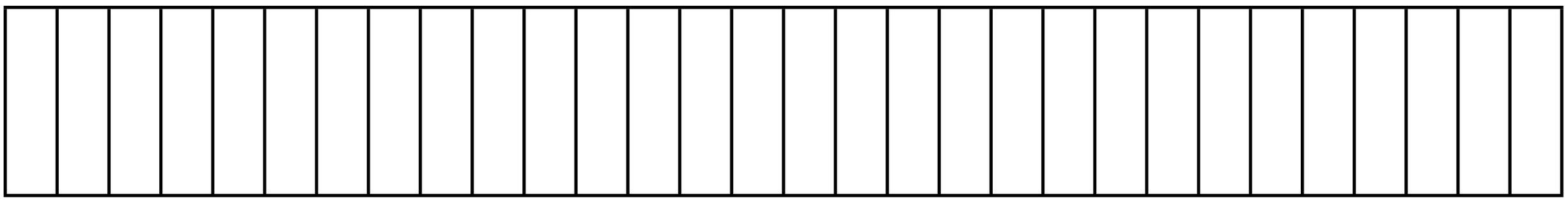
We can also specify how many bits we want to encode a float in - in this case, we round the float to the nearest 1/32nd. This might be plenty good for a health or shield value.

```
writeFlag(1);
writeRangedInt(17, 0, 32);
writeString("Hey");
```



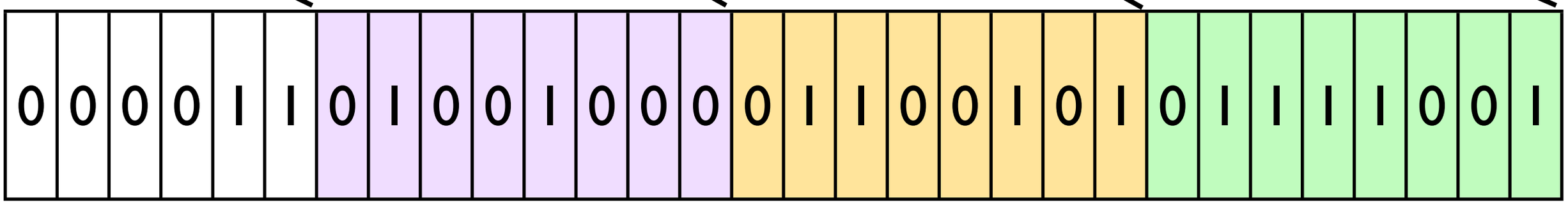
How about something more complex, like a string?

```
writeFlag(1);
writeRangedInt(17, 0, 32);
writeString("Hey");
    writeRangedInt(3, 0, 64);
    writeInt('H', 8); writeInt('e', 8); writeInt('y', 8);
```



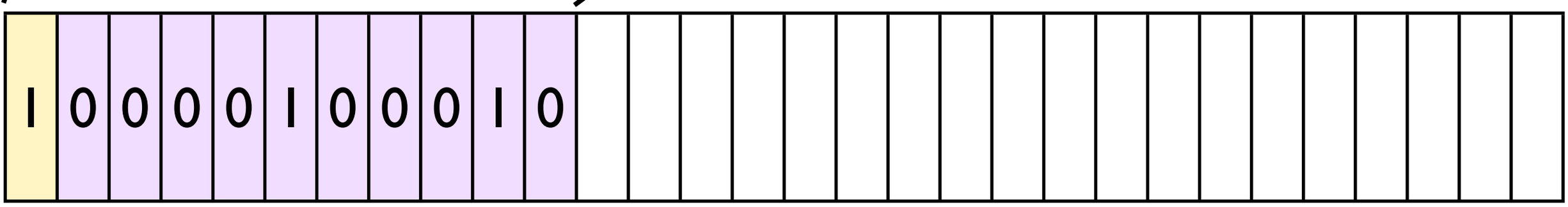
We could write some ints. I skip the length field you'd need, since the slide is of limited width.

```
writeFlag(1);
writeRangedInt(17, 0, 32);
writeString("Hey");
    writeRangedInt(3, 0, 64);
     writeInt('H', 8);    writeInt('e', 8);  writeInt('y', 8);
```



Here we go - 30 bits for the full string. Realistically you'd want more bits on the length field, but I only had so much space. ;)

```
writeFlag(1);
writeRangedInt(17, 0, 32);
writeString("Hey");
   writeFlag(1); writeInt(34, 10);
```



We could also cache commonly used strings in a table, and just reference slots in the table.

Key points to remember:
CPU is cheap compared to the cost of sending network bytes.
Be efficient in encoding data.
Bit pack everything - and only use the bits you need.
Sacrifice precision where you can.

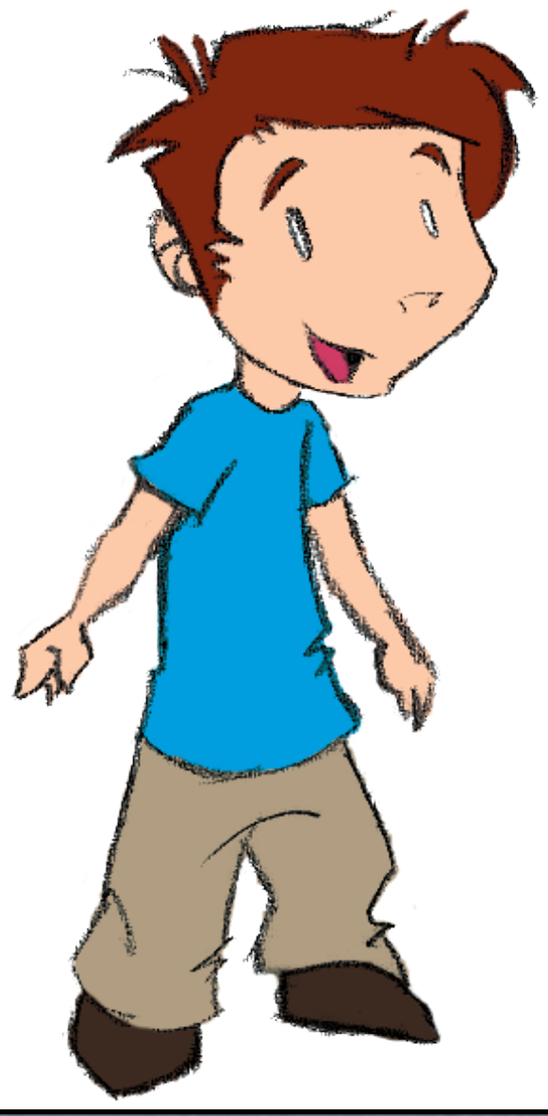
Blockland Brick Example

10,000 bricks in this level, built one by one. Shaving a single bit off of brick update means 1/4 sec faster load. We shaved off 80 bits – that's 20 seconds less time loading, took it from 24 seconds to 4 to load this build.

The overall time isn't a big deal, though, it's the fact that bricks now really fly into place, instead of going slowly as they did before. Perceptually, the load is a lot more satisfying.

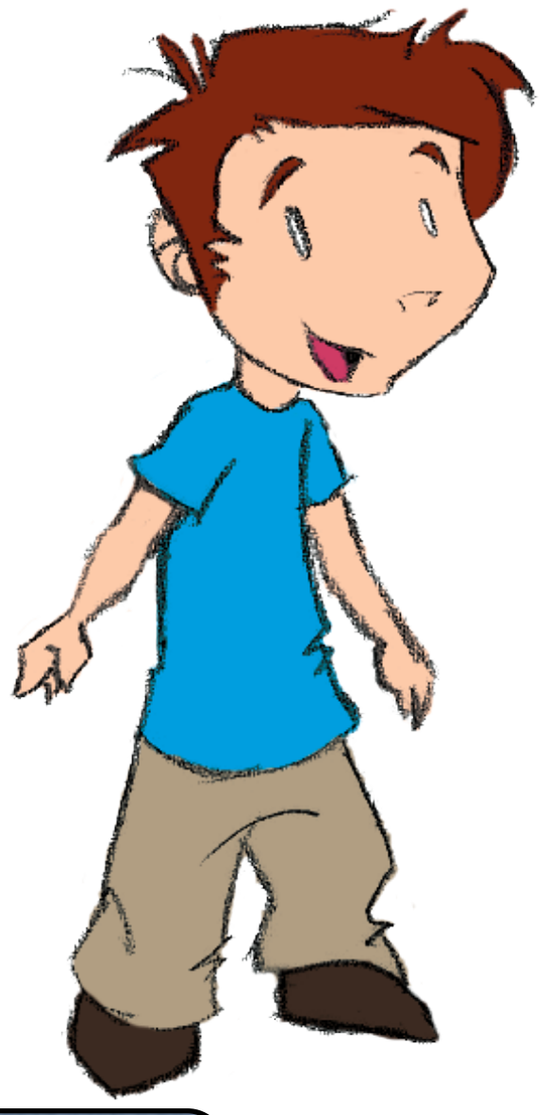
Control Objects

Ghosts are good enough for most game objects - like trees, powerups, lightswitches, flags, and so forth. But in practice they turn out to not work very well for objects that the player directly controls.



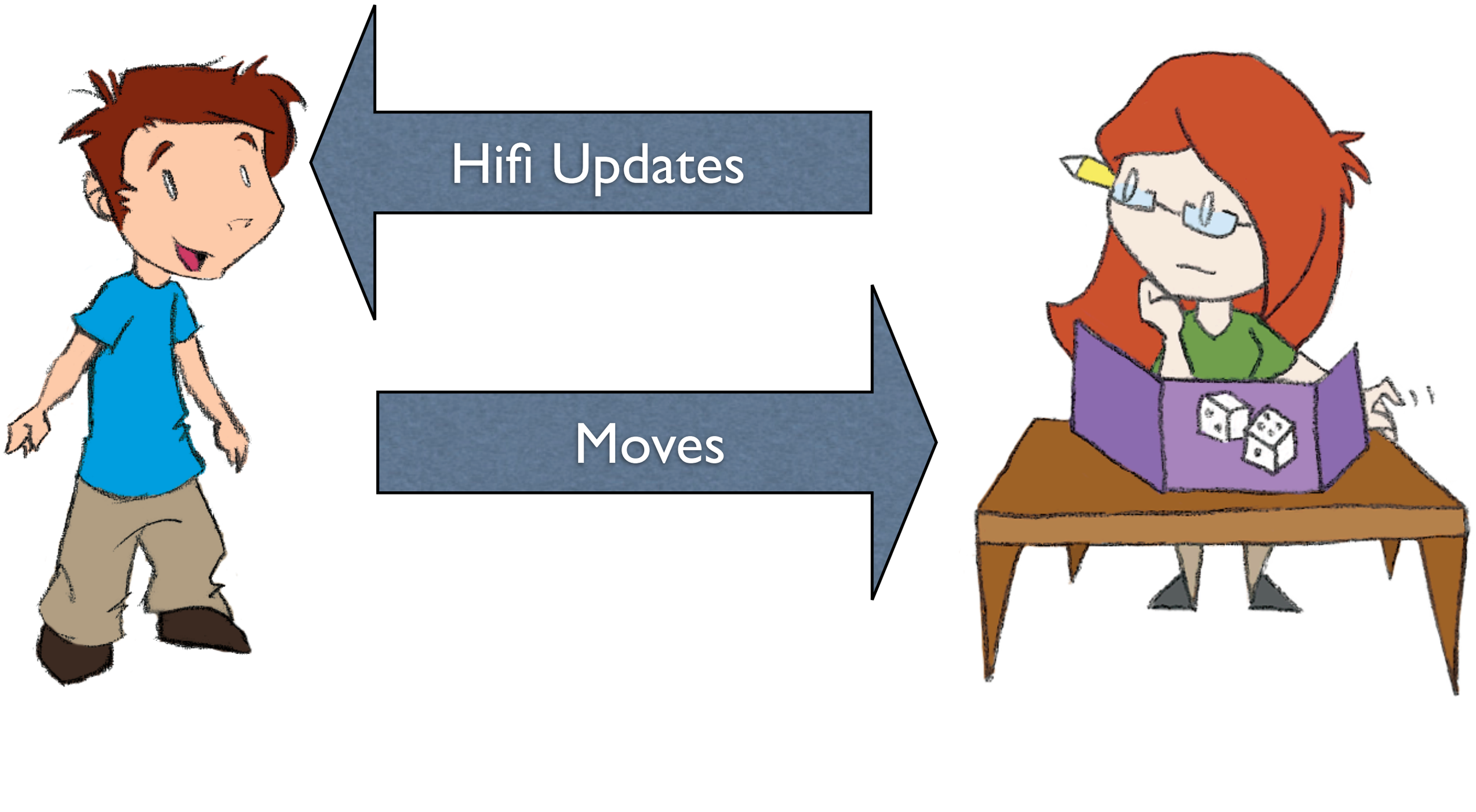
Coming back to our opening example...

Because ghosting is not guaranteed to happen in a specific order or at a specific update rate, it may result in your player moving oddly. Imagine driving a car where the steering wheel only works some of the time. You need immediate feedback all the time.



Coming back to our opening example...

Because ghosting is not guaranteed to happen in a specific order or at a specific update rate, it may result in your player moving oddly. Imagine driving a car where the steering wheel only works some of the time. You need immediate feedback all the time.



- Special piece of networking specifically for the object that that connection is controlling.
- Client sends movement streams - snapshots of input state at different times.
- Server sends high-fidelity updates back - these are full precision, complete state updates sent at the start of every packet. Stay as in-synch as possible!
- Key to keeping control of player/vehicles snapper, synchronized, and responsive.
- Helps prevent cheating (worst case, player can only play optimally but can't break the rules)
- Helps keep everything in synch
- There's some weirdness here; we'll talk about this more later.

Moves are based on sampling input every so many milliseconds.

The client packs the N most recent moves into the packets it sends to the server. As the server indicates it has received packets (using the notify protocol), we remove them from the pending move queue.

As you can see in this example, this means some moves are sent more than once - we're trading bandwidth for immediacy and reliability in this case. Because moves are always in-flight to the server, it's always getting decent move info.

We also send a CRC of the move object's state. This is important for the server. It only sends a full state update for the control object if the CRC doesn't match its own copy of the control object...

Packet 100
CRC
Move 17
Move 18
Move 19
Move 20
Move 21

Moves are based on sampling input every so many milliseconds.

The client packs the N most recent moves into the packets it sends to the server. As the server indicates it has received packets (using the notify protocol), we remove them from the pending move queue.

As you can see in this example, this means some moves are sent more than once - we're trading bandwidth for immediacy and reliability in this case. Because moves are always in-flight to the server, it's always getting decent move info.

We also send a CRC of the move object's state. This is important for the server. It only sends a full state update for the control object if the CRC doesn't match its own copy of the control object...

Packet 101
CRC
Move 20
Move 21
Move 22
Move 23
Move 24

Packet 100
CRC
Move 17
Move 18
Move 19
Move 20
Move 21

Moves are based on sampling input every so many milliseconds.

The client packs the N most recent moves into the packets it sends to the server. As the server indicates it has received packets (using the notify protocol), we remove them from the pending move queue.

As you can see in this example, this means some moves are sent more than once - we're trading bandwidth for immediacy and reliability in this case. Because moves are always in-flight to the server, it's always getting decent move info.

We also send a CRC of the move object's state. This is important for the server. It only sends a full state update for the control object if the CRC doesn't match its own copy of the control object...

Packet 102
CRC
Move 24
Move 25
Move 26
Move 27
Move 28

Packet 101
CRC
Move 20
Move 21
Move 22
Move 23
Move 24

Packet 100
CRC
Move 17
Move 18
Move 19
Move 20
Move 21

Moves are based on sampling input every so many milliseconds.

The client packs the N most recent moves into the packets it sends to the server. As the server indicates it has received packets (using the notify protocol), we remove them from the pending move queue.

As you can see in this example, this means some moves are sent more than once - we're trading bandwidth for immediacy and reliability in this case. Because moves are always in-flight to the server, it's always getting decent move info.

We also send a CRC of the move object's state. This is important for the server. It only sends a full state update for the control object if the CRC doesn't match its own copy of the control object...

Packet 103
CRC
Move 27
Move 28
Move 29
Move 30
Move 31

Packet 102
CRC
Move 24
Move 25
Move 26
Move 27
Move 28

Packet 101
CRC
Move 20
Move 21
Move 22
Move 23
Move 24

Packet 100
CRC
Move 17
Move 18
Move 19
Move 20
Move 21

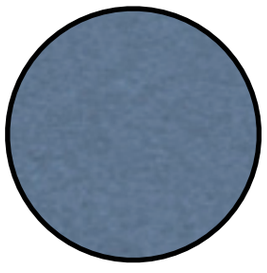
Moves are based on sampling input every so many milliseconds.

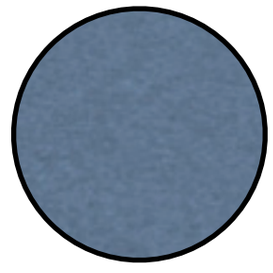
The client packs the N most recent moves into the packets it sends to the server. As the server indicates it has received packets (using the notify protocol), we remove them from the pending move queue.

As you can see in this example, this means some moves are sent more than once - we're trading bandwidth for immediacy and reliability in this case. Because moves are always in-flight to the server, it's always getting decent move info.

We also send a CRC of the move object's state. This is important for the server. It only sends a full state update for the control object if the CRC doesn't match its own copy of the control object...

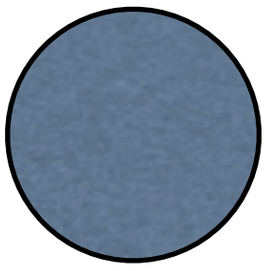
Client



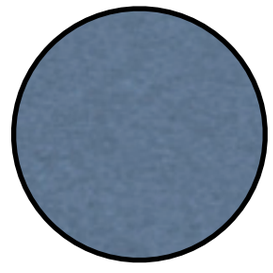


Server

What about that CRC field? Well, every packet the client sends encodes enough info about the control object that the server can check if things match. If they do it doesn’t send an update. Like in this case....

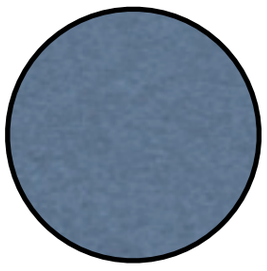


Client

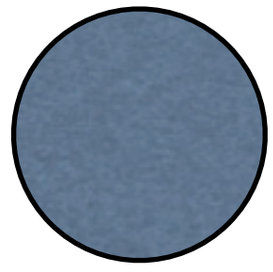


Server

What about that CRC field? Well, every packet the client sends encodes enough info about the control object that the server can check if things match. If they do it doesn’t send an update. Like in this case....



Client



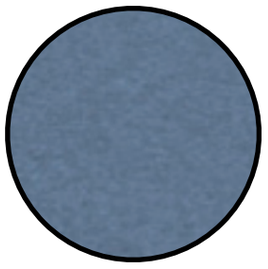
Server

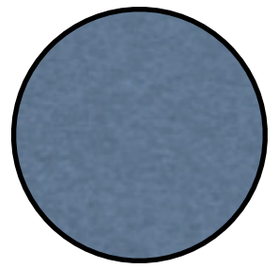
What about that CRC field? Well, every packet the client sends encodes enough info about the control object that the server can check if things match. If they do it doesn’t send an update. Like in this case....

Client

Server

But suppose a collision occurs and the client has gotten that ghost yet – the CRCs will mismatch and we'll know we need to send an update.

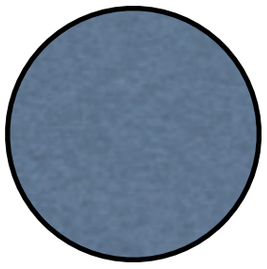


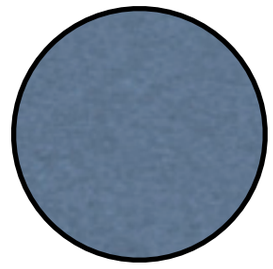


Client

Server

But suppose a collision occurs and the client has gotten that ghost yet – the CRCs will mismatch and we'll know we need to send an update.

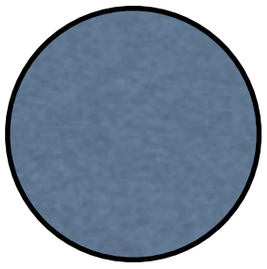


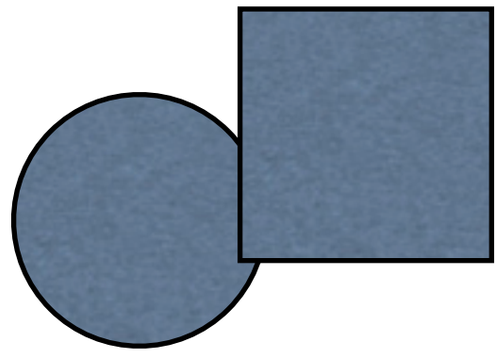


Client

Server

But suppose a collision occurs and the client has gotten that ghost yet – the CRCs will mismatch and we'll know we need to send an update.

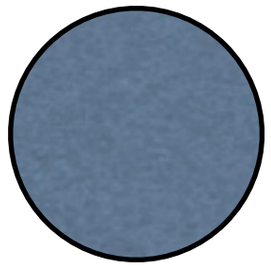




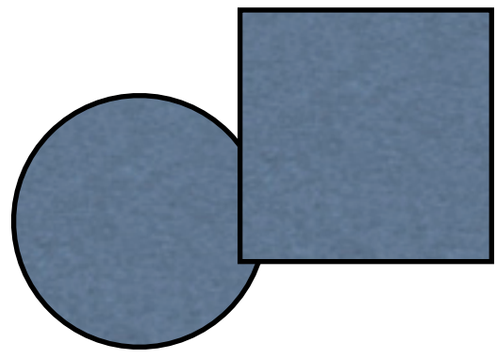
Client

Server

But suppose a collision occurs and the client has gotten that ghost yet – the CRCs will mismatch and we'll know we need to send an update.



CRC Mismatch!



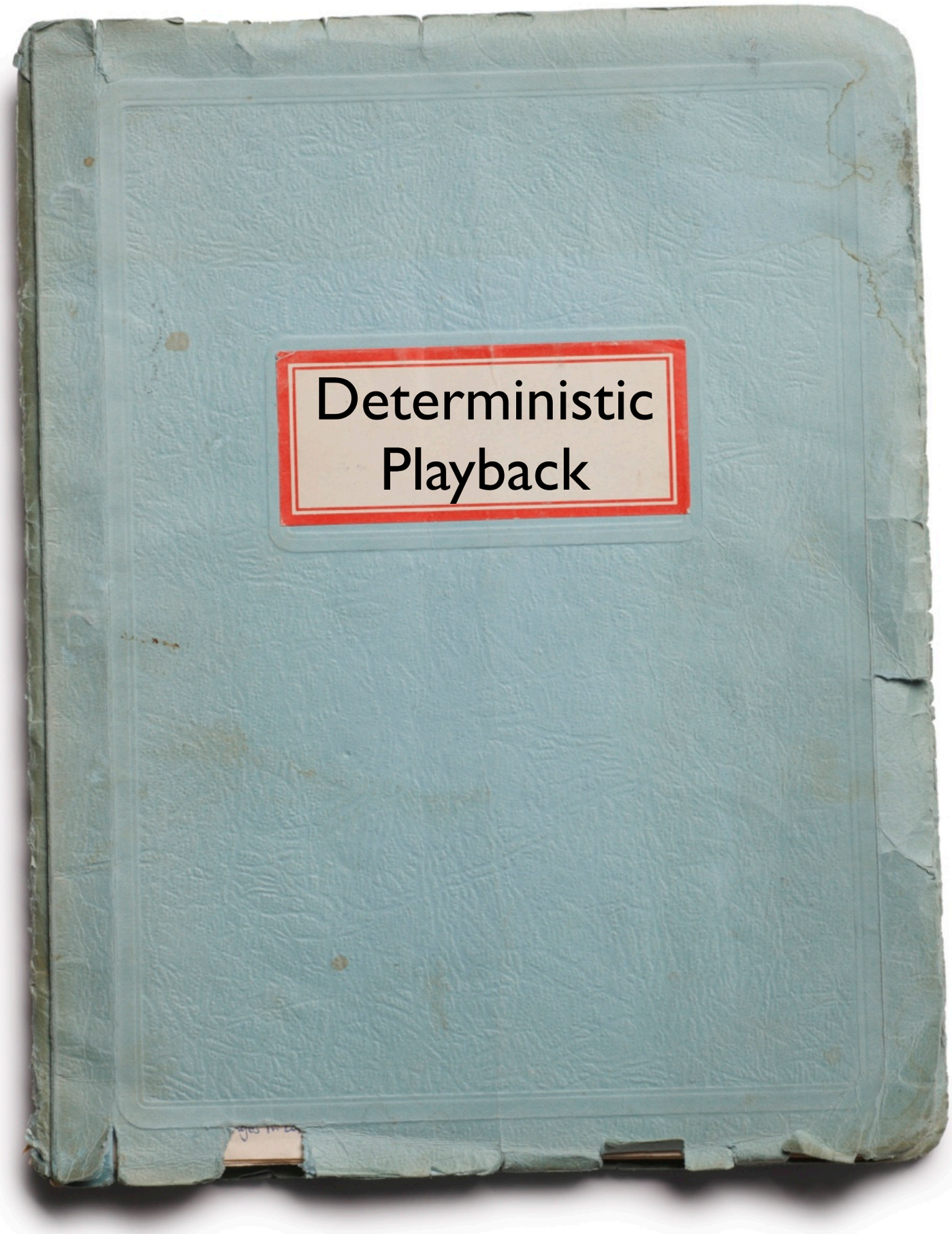
Client

Server

But suppose a collision occurs and the client has gotten that ghost yet – the CRCs will mismatch and we'll know we need to send an update.

Networking Your Simulation

- The transport aspect is important.
- But your simulation must support networking from the ground up for best results.
- There are a couple pieces to this...



The very first thing you should do is add a journalling system that will let you do deterministic playback of game sessions. If your game can't run deterministically, networking will be nigh impossible.

For journalling: Cannot overemphasize this. (Zap testing anecdote.)

Consistency

http://flickr.com/photos/wsdot/820302116/ – WSDOT photo by Jim Culp.

This is an expansion joint on the (new) Tacoma Narrows Bridge.

Consistency is like concrete. If you try to be 100% consistent there will be cracks everywhere. But put a few places in that don't need to be consistent, and everything works out a lot better.

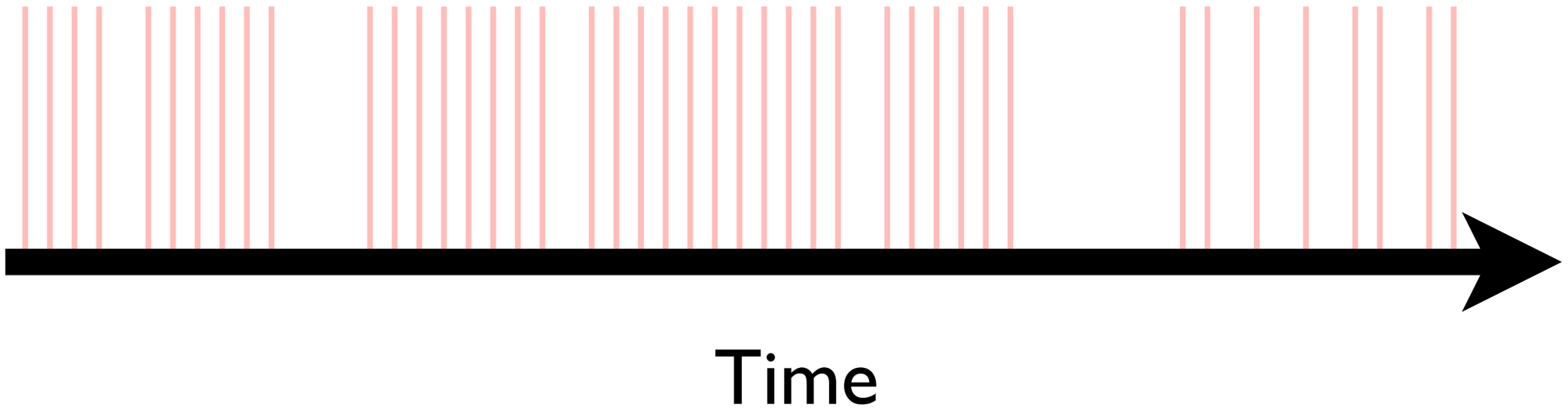
In other words – you only have a certain amount of consistency. Spend it wisely. You can get a lot of mileage out of client-side effects that don't require any networking.

- You will never achieve 100% consistency.
- Speed of light prevents this if nothing else. <stuartchesire.org/rants/Latency.html>
- But you can do various things to give the illusion of consistency.

This is time in our app...

Here are the frames. These are the actual opportunities to process/render that we have.

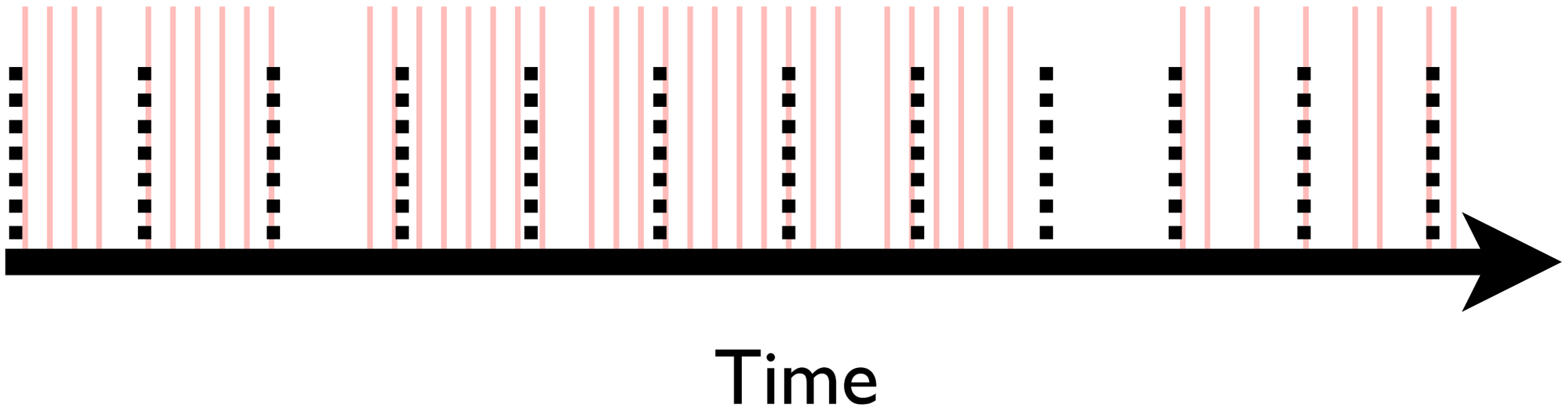
Here are the ticks, where we updated game state. These are guaranteed to happen. I usually do ticks at 32Hz but you may change that. Having fixed ticks helps a ton in networking, as it helps keep things deterministic.



This is time in our app...

Here are the frames. These are the actual opportunities to process/render that we have.

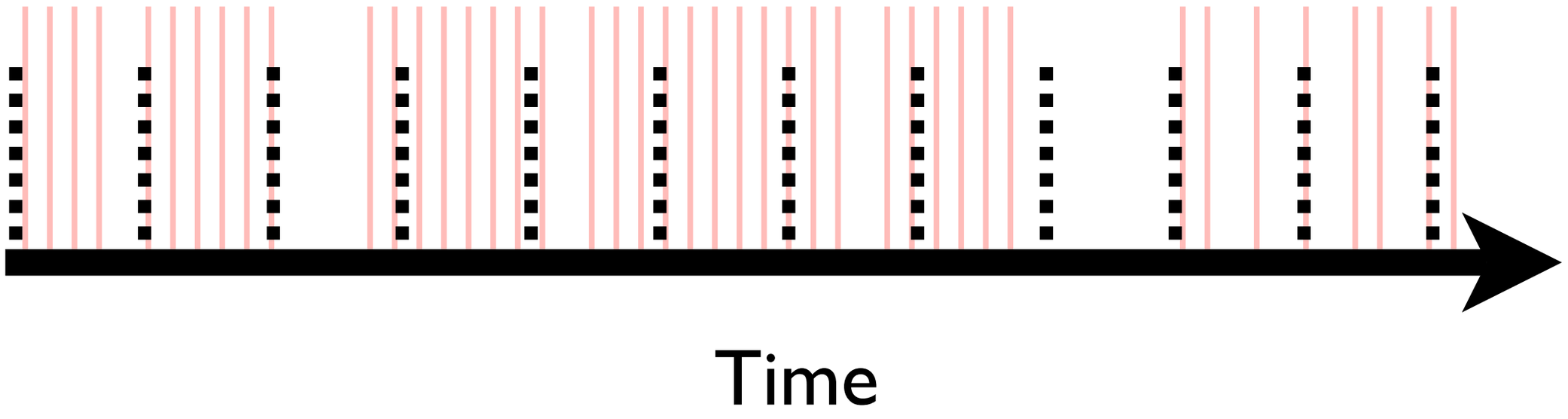
Here are the ticks, where we updated game state. These are guaranteed to happen. I usually do ticks at 32Hz but you may change that. Having fixed ticks helps a ton in networking, as it helps keep things deterministic.



This is time in our app...

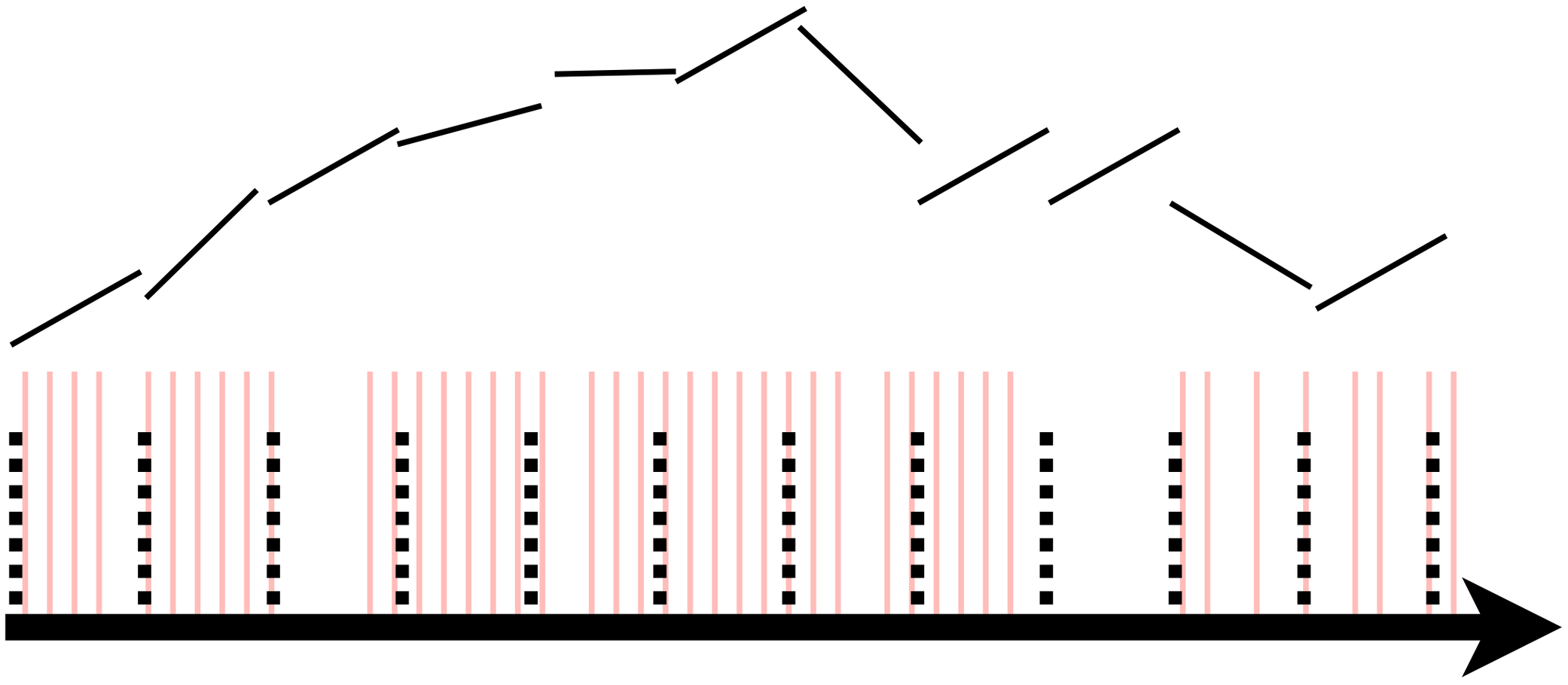
Here are the frames. These are the actual opportunities to process/render that we have.

Here are the ticks, where we updated game state. These are guaranteed to happen. I usually do ticks at 32Hz but you may change that. Having fixed ticks helps a ton in networking, as it helps keep things deterministic.



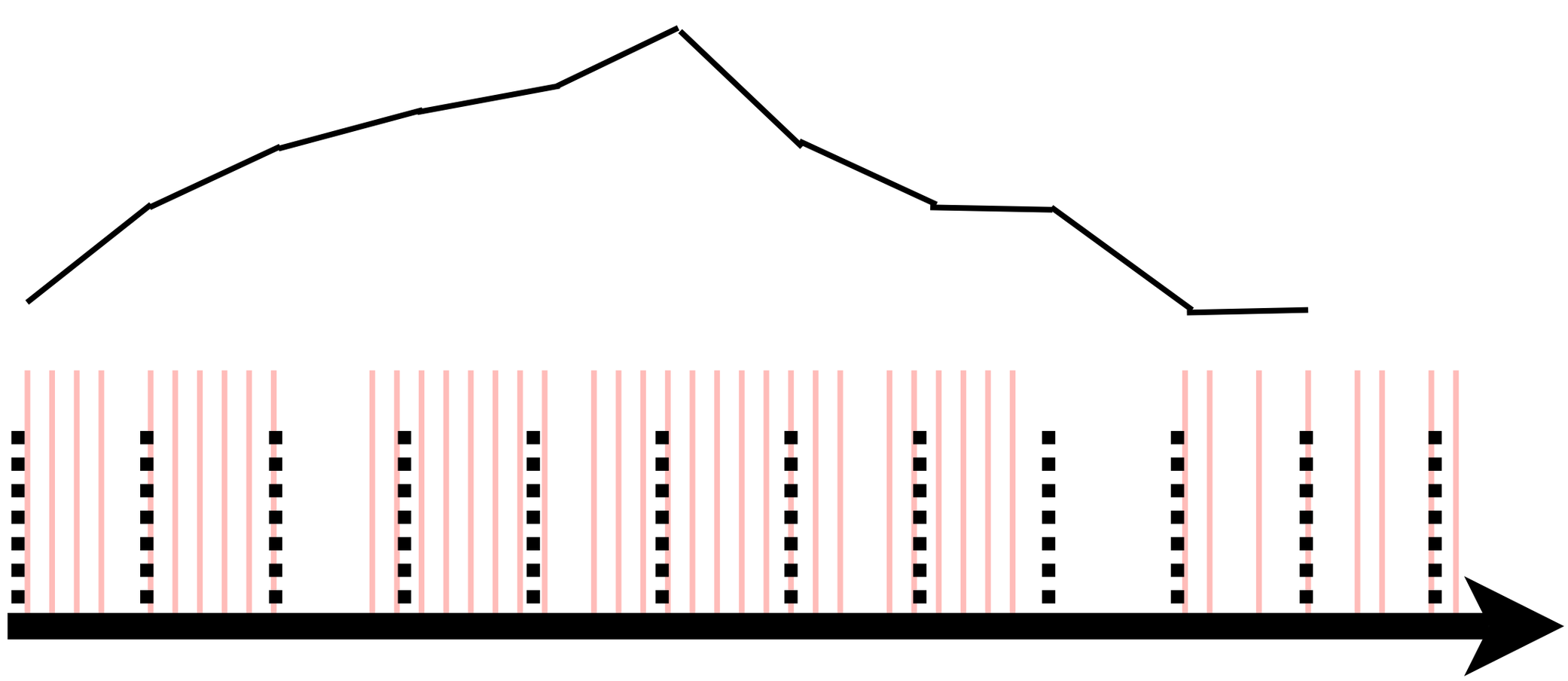
You want to interpolate to known values rather than predict. This adds a fixed amount of lag, but that's ok - people readily get used to constant lag. Most of our nervous system has fixed lag in it.

Why? Well, here's what it looks like if you try to predict a value that's getting updated from an external source. You're wrong almost all the time, even if only by a little bit.

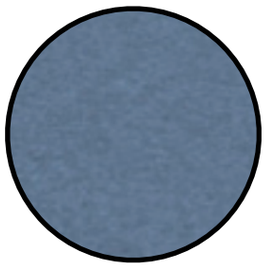


You want to interpolate to known values rather than predict. This adds a fixed amount of lag, but that's ok - people readily get used to constant lag. Most of our nervous system has fixed lag in it.

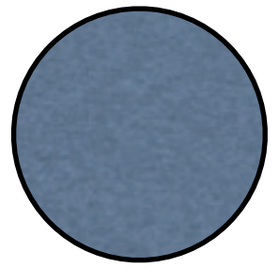
Why? Well, here's what it looks like if you try to predict a value that's getting updated from an external source. You're wrong almost all the time, even if only by a little bit.



This is how it SHOULD look. And if you interpolate towards known values, you get this result with no extra work.



Client

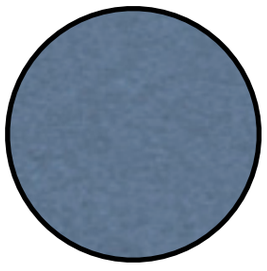


Server

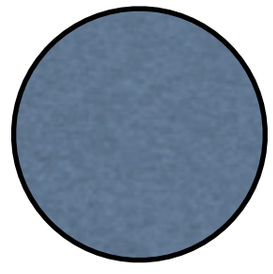
Lumpy time - problem that comes up with move streams.

Tick - no packet. Tick - no packet. Tick - no packet.

Suddenly, the server gets the moves and catches up!



Client

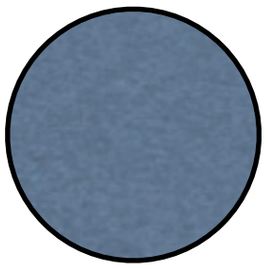


Server

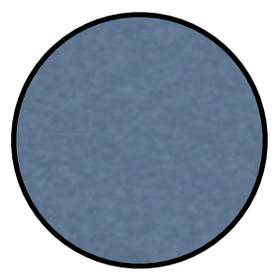
Lumpy time - problem that comes up with move streams.

Tick - no packet. Tick - no packet. Tick - no packet.

Suddenly, the server gets the moves and catches up!



Client

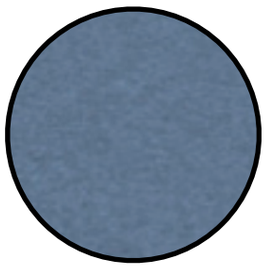


Server

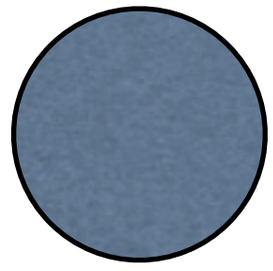
Lumpy time - problem that comes up with move streams.

Tick - no packet. Tick - no packet. Tick - no packet.

Suddenly, the server gets the moves and catches up!



Client

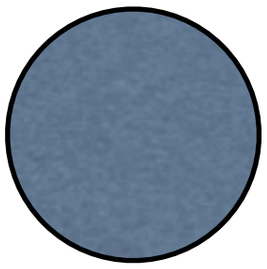


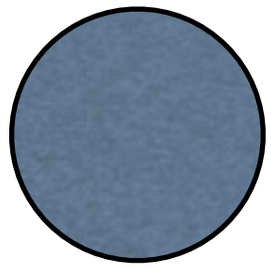
Server

Lumpy time - problem that comes up with move streams.

Tick - no packet. Tick - no packet. Tick - no packet.

Suddenly, the server gets the moves and catches up!





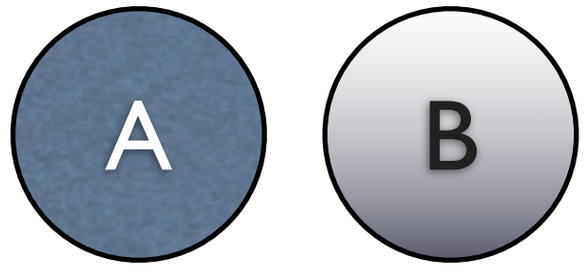
Client

Server

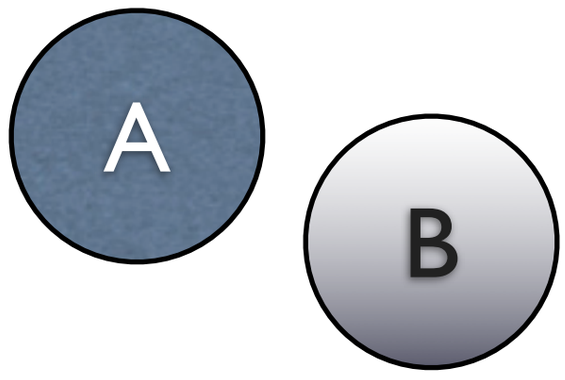
Lumpy time - problem that comes up with move streams.

Tick - no packet. Tick - no packet. Tick - no packet.

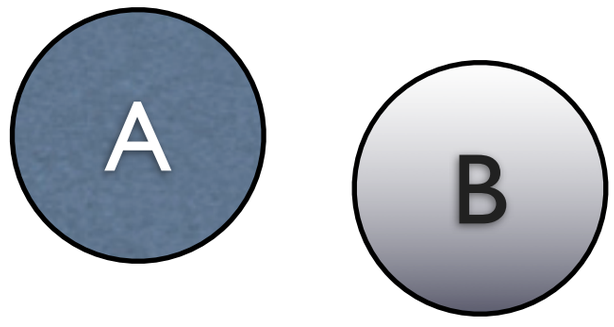
Suddenly, the server gets the moves and catches up!

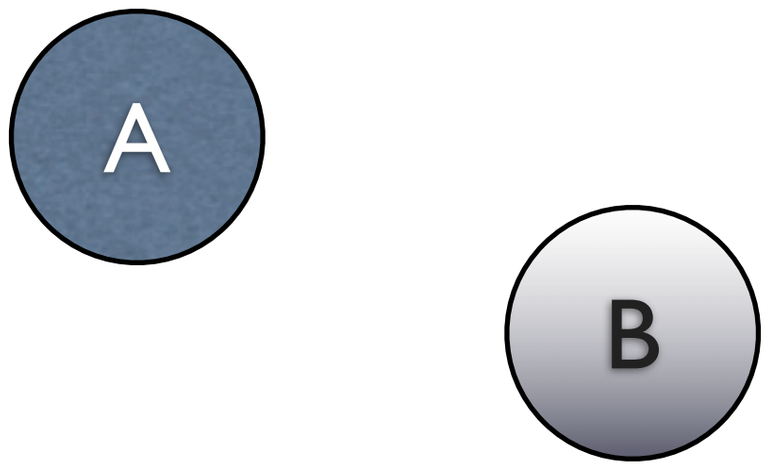


How can this go terribly wrong? Well, check this out. We have a little FPS.

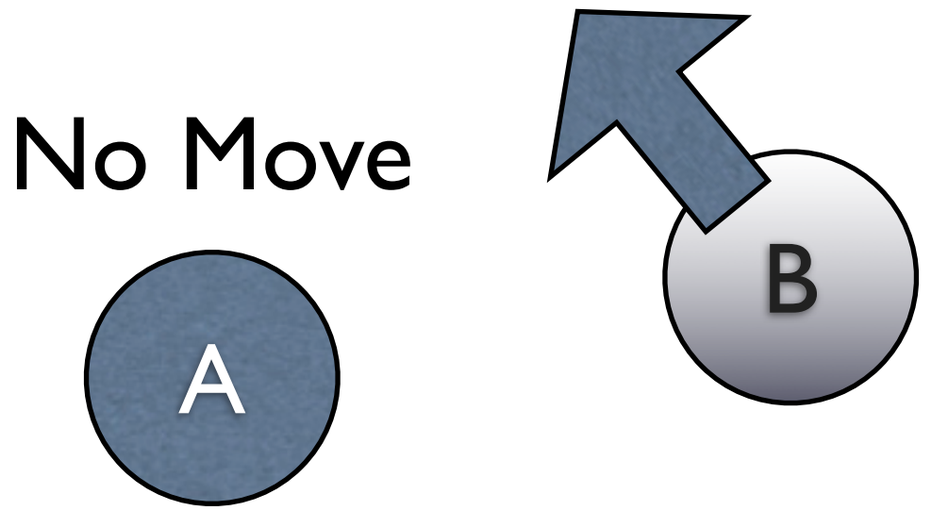


Players are running... tick tick tick

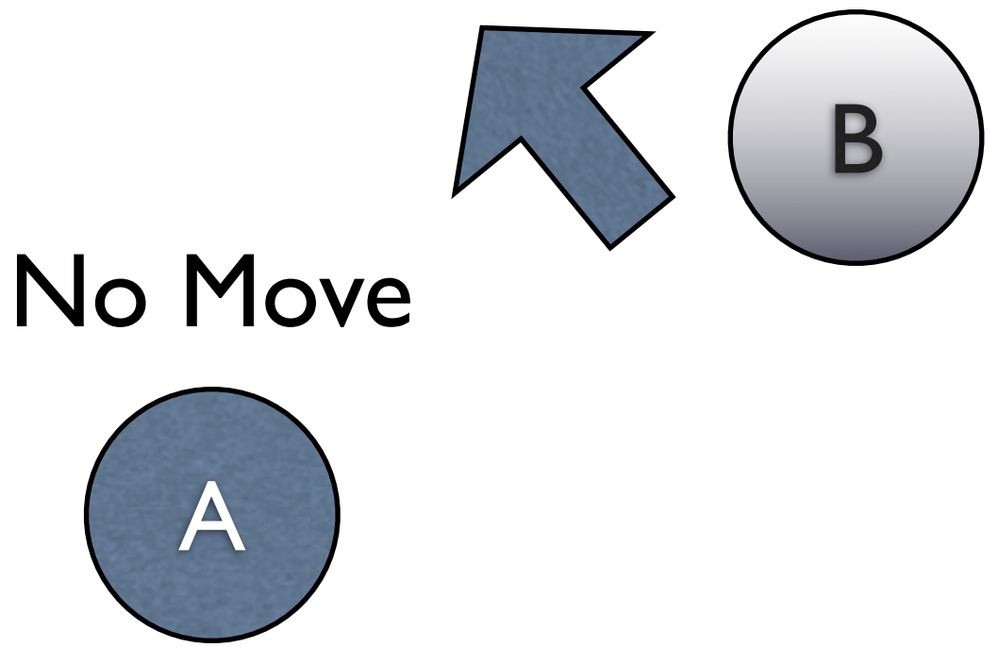

A
B



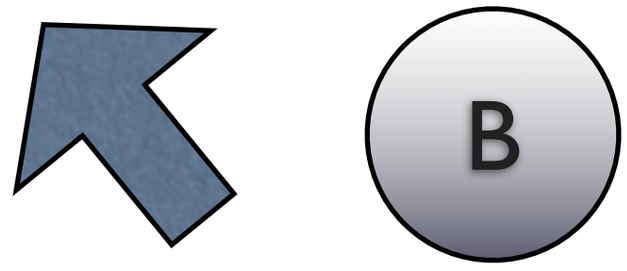
Everything goes smooth, until



Bam! A starts dropping packets. Maybe this is an honest problem, or maybe he unplugged his router.



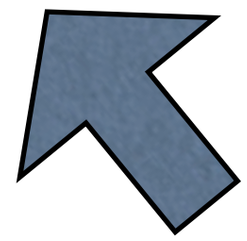
B and the rocket move normally...



B

No Move

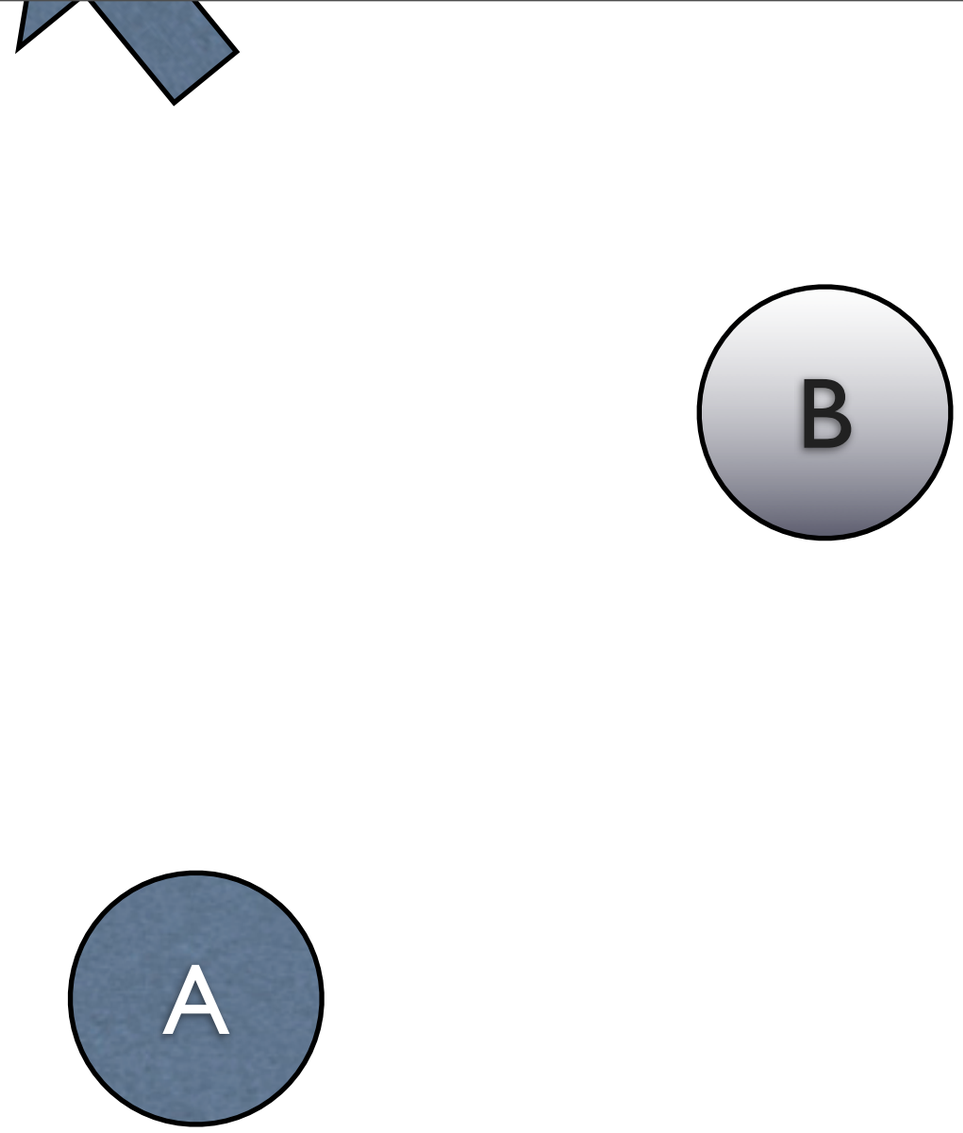
A



B

No Move

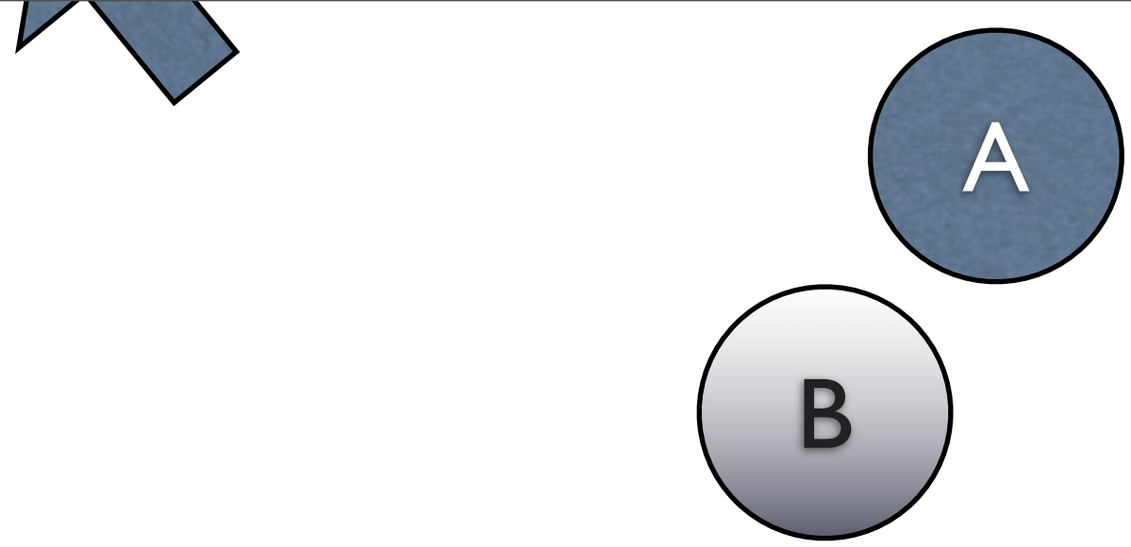
A



Now the rocket is out of the way... and A’s packet shows up with a ton of moves. Whoosh!

He avoided the rocket.

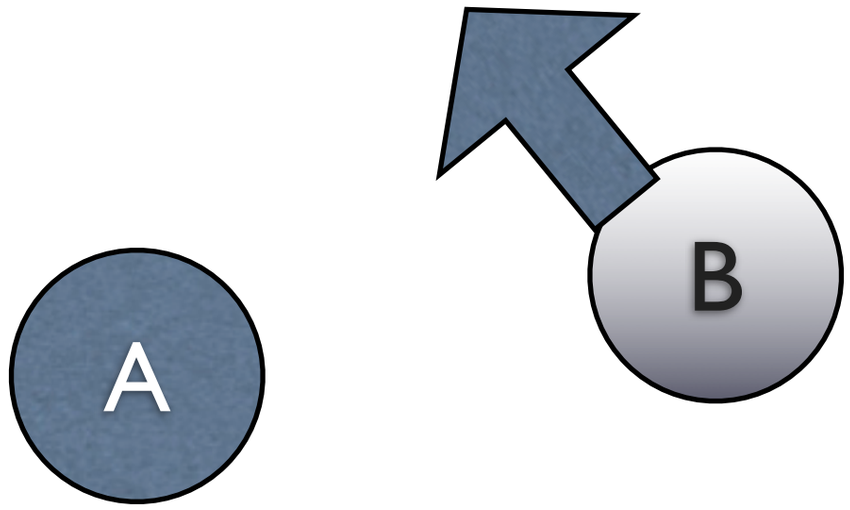
How can we fix this? Let’s rewind time a little.



Now the rocket is out of the way... and A's packet shows up with a ton of moves. Whoosh!

He avoided the rocket.

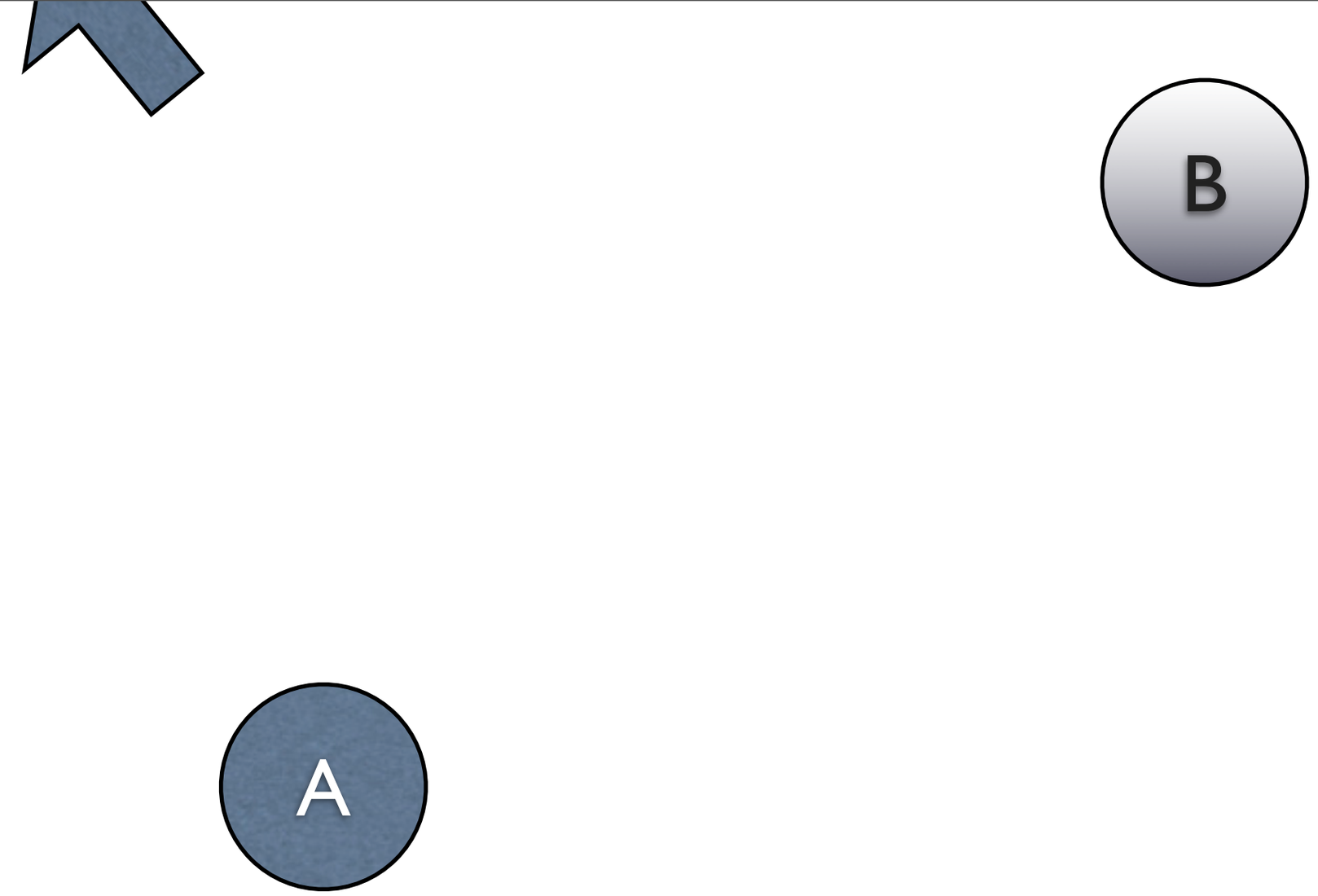
How can we fix this? Let's rewind time a little.



Not every game needs this solution, btw. For instance, Zap and all Torque games don't use it at all and get along OK. But it's important for certain genres, like racing games. Source does this as well.

What you do is keep a history of the past N states for game objects, based on how long you let a game object wait for user moves before simulating without them.

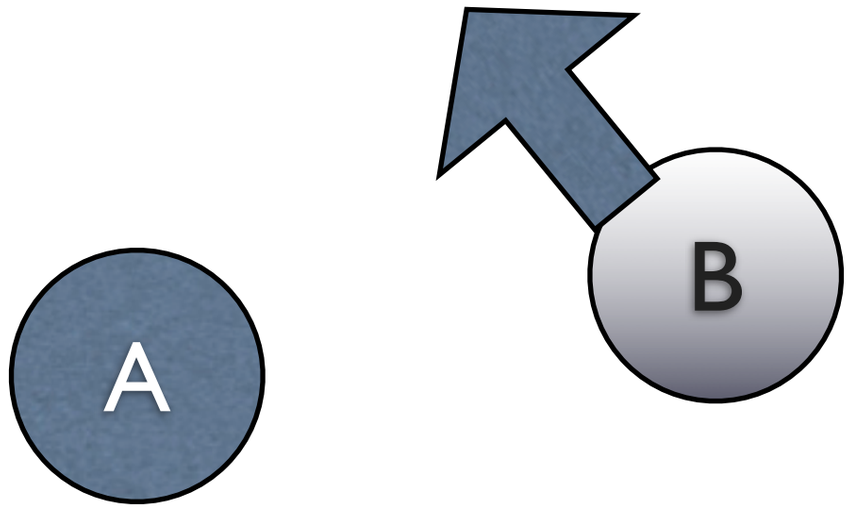
So we tick ahead without moves for A.. then when A's moves come in...



Not every game needs this solution, btw. For instance, Zap and all Torque games don’t use it at all and get along OK. But it’s important for certain genres, like racing games. Source does this as well.

What you do is keep a history of the past N states for game objects, based on how long you let a game object wait for user moves before simulating without them.

So we tick ahead without moves for A.. then when A’s moves come in...



We rewind and resimulate. Collisions happen as they should and everything works.

This can be very costly, though. We found it was helpful to only rewind nearby objects than the entire world state. We also implemented a cache to help reduce overhead of calculating object state.

We rewind and resimulate. Collisions happen as they should and everything works.

This can be very costly, though. We found it was helpful to only rewind nearby objects than the entire world state. We also implemented a cache to help reduce overhead of calculating object state.

Details

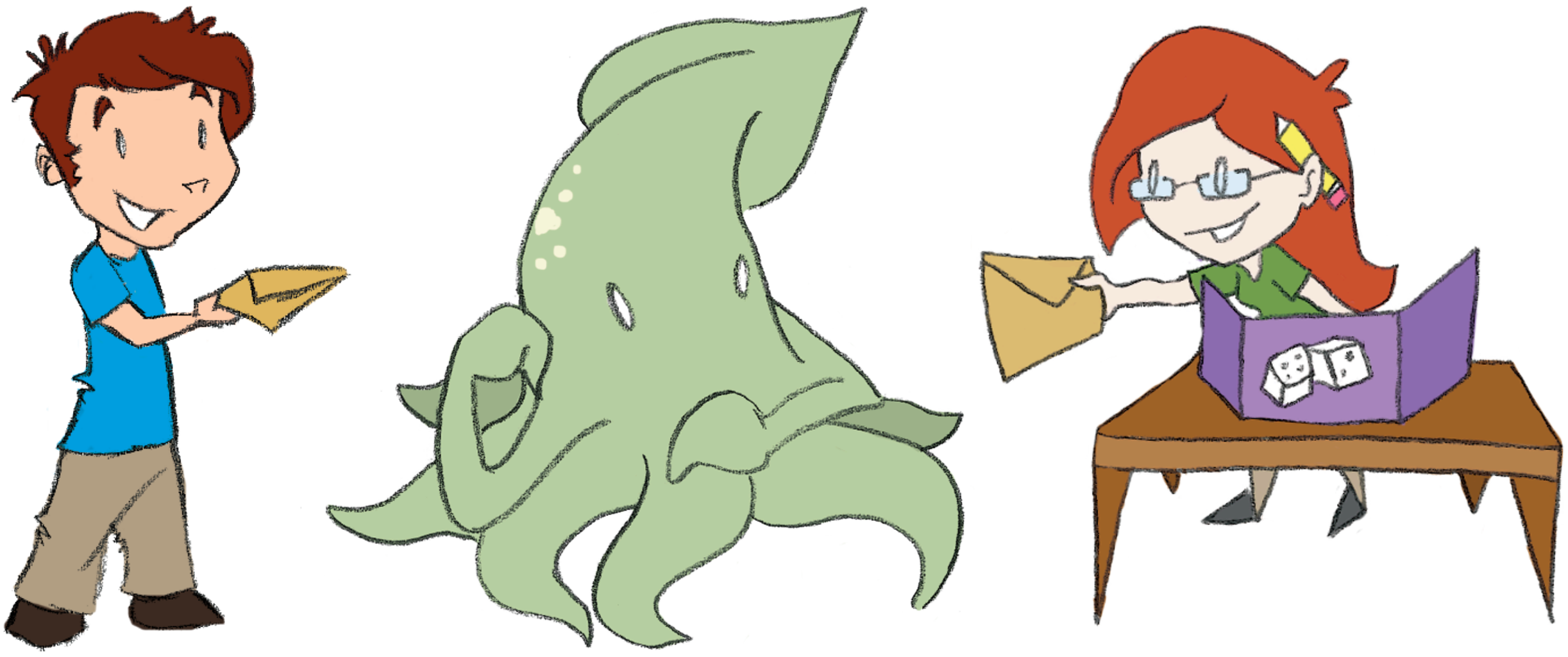
Good networking is about getting all the details right. The difference between a good networking system used poorly and a bad networking system used well isn't much - they both suck. You have to have a good system and use it right.

Packets

	Fixed Size	Variable Size
Fixed Rate	Action Game	
Variable Rate		Application

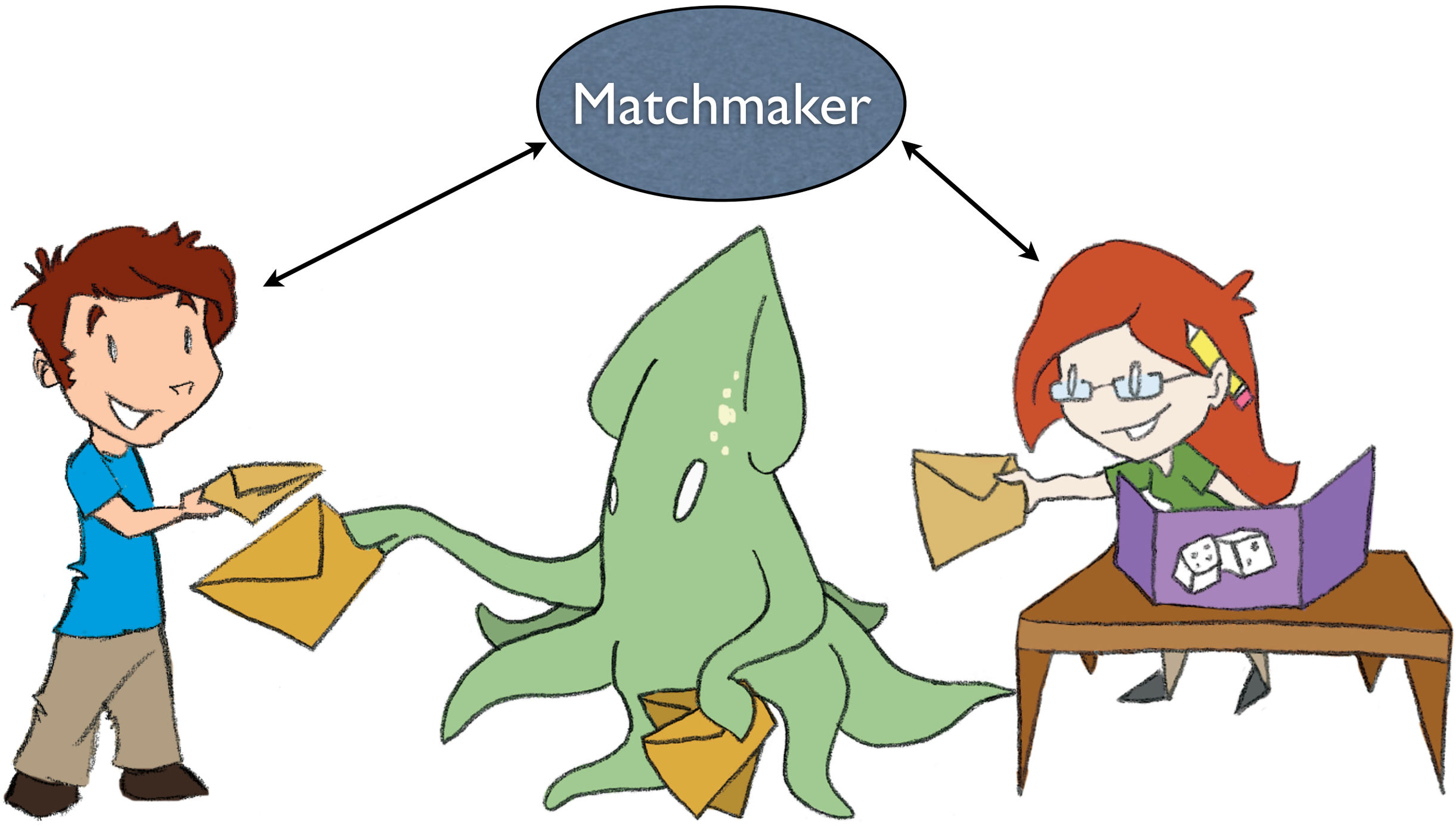
This depends on the situation...

NAT Traversal



Sometimes one or both parties in a connection will be behind a firewall or uncooperative router.

NAT Traversal



If you have a matchmaker service that can help “punch” through, you can get formerly unaccessible players out into the world.

The master server in OpenTNL does this. I’ll also post a link to the paper, if I can find it.

Inter-Language Support

- Document the protocol.
- Remove the opaque, build-dependent IDs.
- Voila - communicate between different languages.

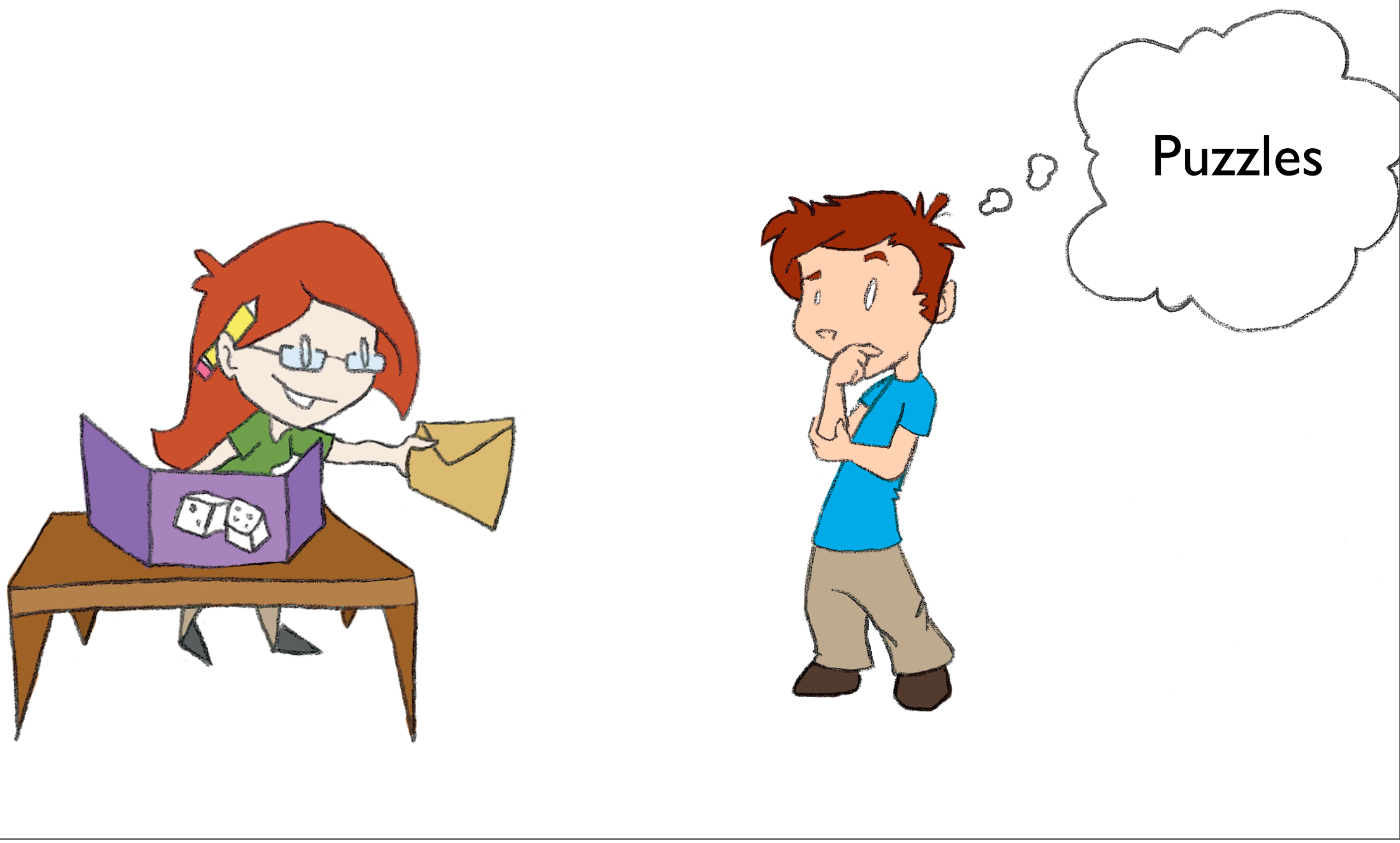
Use TCP

Why use TCP? (click) Some languages don’t support UDP.

Use TCP

Why use TCP? (click) Some languages don’t support UDP.

Preventing DoS



You can use computational puzzles to help prevent denial of service attacks. The server can still have its connection saturated, but attackers can't consume resources.

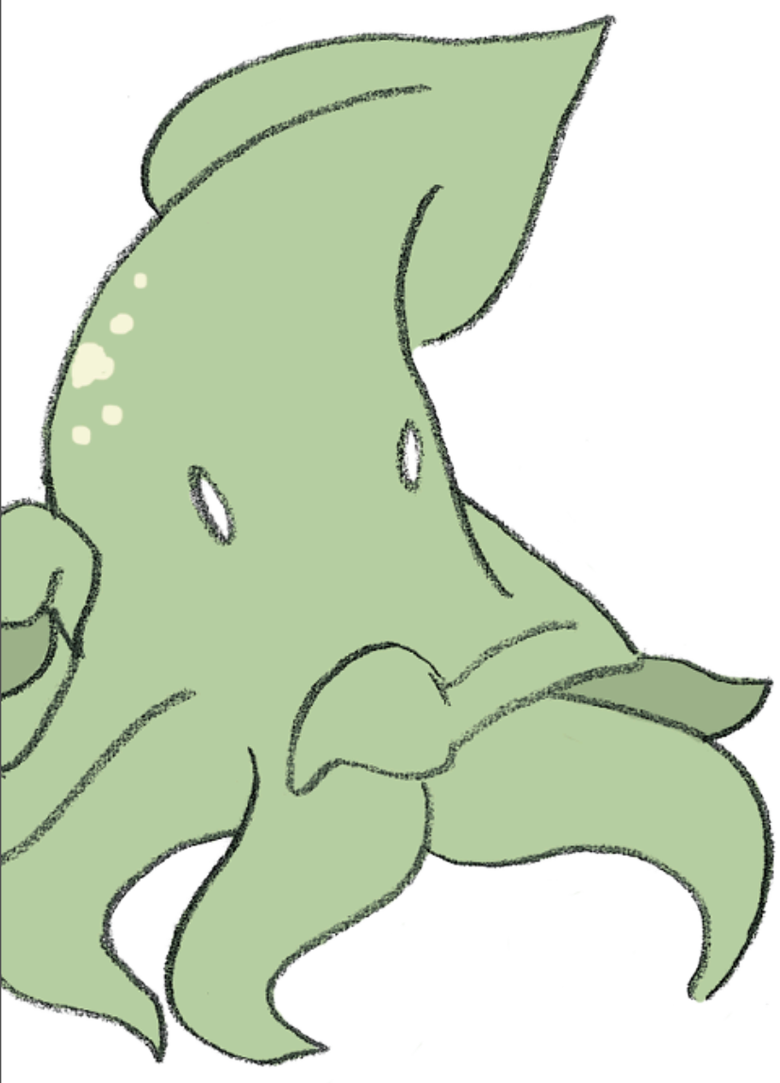
A puzzle is a specific computational problem that is cheap to generate, but costly to solve. The difficulty can also be adjusted. The server can easily outpace clients, and make harder problems == longer to solve if lots of people are trying to connect. The puzzle is rotated at regular intervals.

With careful implementation, you can avoid allocating a single byte when a connection request is made, until the client presents a solved puzzle.

Encryption

- Symmetric cipher with key exchange.
- Signing of packets to ensure integrity.
- Lots of useful possibilities.

Conclusion



- Only send the bits you have to.
- Embrace latency and packet loss.
- Make sure you can debug.
- The devil is in the details.

Don’t forget to turn in your evaluation forms.

Questions?

Contact Me

ben.garney@gmail.com

Download These Slides

http://coderhump.com/austin08/