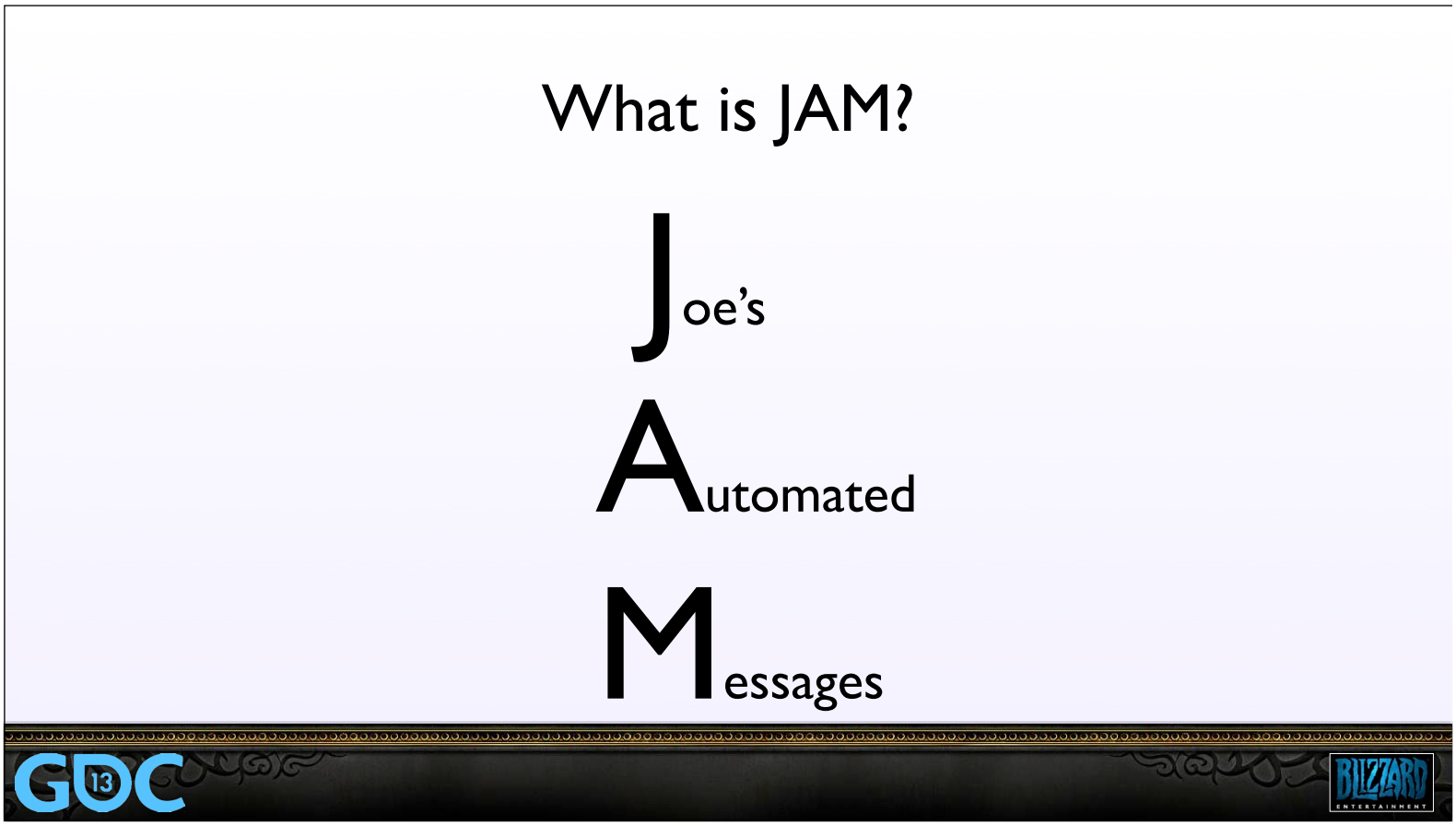
Network Serialization and Routing in World of Warcraft

Joe Rumsey
jrumsey@blizzard.com
Twitter: @joerumz

Cell phones
Email surveys

My name is Joe Rumsey. I'm a principal software engineer. Means when programmers get in trouble they get sent to my office. I've been at Blizzard for the last 12 years. I was the lead server engineer on World of Warcraft, and today I'm working on Blizzard's next MMO. Today I'm going to show you JAM, which is the application level network library for World of Warcraft and our next MMO.



Yes, I named it after myself. I'm sorry, it was an accident. JAM is a network serialization and transport library used by Blizzard's MMOs. There are some similar projects that are widely available today, such as Google Protocol Buffers, but JAM development dates back to 2001, when there weren't any good alternatives. Today, JAM stacks up pretty well against those alternatives.

I'll outline the why and how of JAM and why we're still happily using it today. Hopefully, some of you will be inspired to create something similar for your games. Or come work for us, we're always hiring! At the very least, I hope if you're not using something like JAM already, you will decide to check out protobufs, Thrift, or another similar tech.

The Problem

Game servers need to communicate with each other

When writing an MMO, it's a given that there will be multiple servers, and types of servers, involved. All of these servers need to talk to each other. The resulting system can be quite complex, so we need systems that let us manage that complexity as simply as possible. I'm going to demonstrate the core systems that World of Warcraft uses to mitigate that complexity.

We'll see code that automatically serializes both simple structures of flat data, as well as dynamic data types, code that tracks and manages connections for us, and does all of the above easily and efficiently. At the time WoW shipped in 2004, JAM was only used between game servers. Today it is also used with internal tools and in current Blizzard projects it is used between game clients and servers as well, but for today's talk I am only going to talk about inter-server communication.

Manual serialization is error-prone

```
void Serialize(stream &msg)
{
    vector<int> values;
    // ...Fill in some values...
    msg << values.size();
    for(int i = values.size(); --i;)
    {
        msg << values[i];
    }
}
```

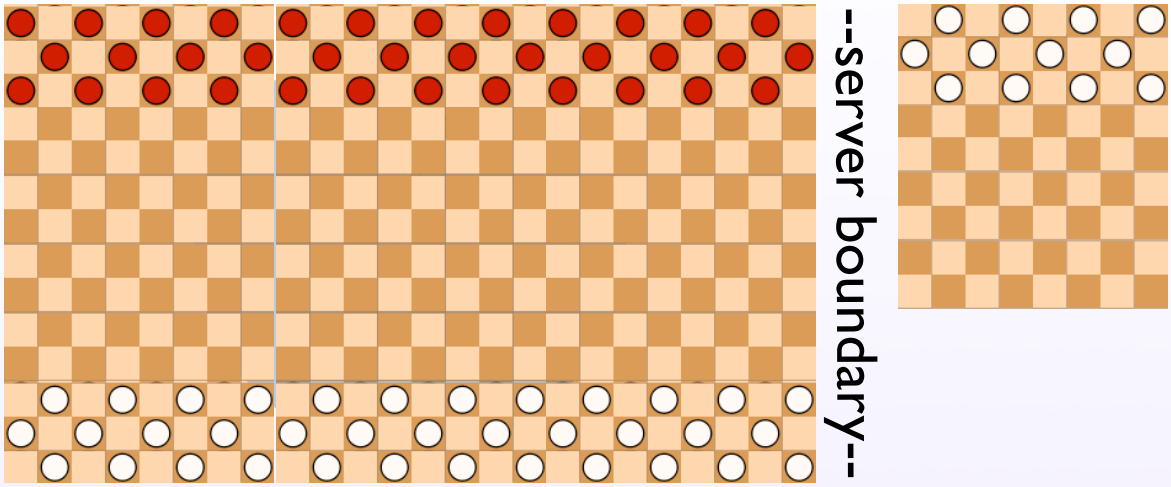
```
void Deserialize(stream &msg)
{
    vector<int> values;
    int size;
    msg >> size;
    values.resize(size);
    for(int i = size; i--;)
    {
        msg >> values[i];
    }
}
```

Many of you probably serialize your network messages manually. If it was good enough twenty years ago, it's still good enough today, right? This is what your code might look like without JAM or a library like it. For anything you want to send and receive, you have to write both ends of that code. This code has a bug, have you spotted it?

[ADVANCE] The loop iterator has an off by one. You might have spotted that easily, but if it was buried in a more complex message and these were in two separate files you might have had a much harder time. You shouldn't have to spend your time looking for things like this! If it's subtle enough, you might not find this error until your game went live. More likely, you will spend hours debugging your message passing when you could have been making your game awesome. Programming errors exactly like this were the very first thing JAM was created to avoid.

Manual serialization doesn't scale

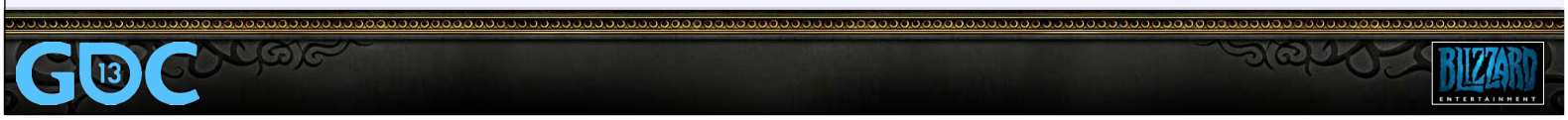
World Of Checkers



If you're writing a chat server, you can probably get away with serializing your data manually. If it's a checkers game with chat, it still probably won't bite you in the ass. And by the way, even though PR told me no way, I said "screw that, I gotta give 'em real code at GDC!", so I'm announcing it right here, right now, to you guys first. Blizzard's next-gen MMO
[ADVANCE] is actually World Of Checkers!
So our checkers MMO has has thousands of pieces, and each of those pieces has dozens of attributes.
[ADVANCE]Not all the pieces can see all the other pieces.
[ADVANCE]Parts of the board are even on different servers. Pieces can be added and removed dynamically. Some pieces have scripts that control their behavior.
[ADVANCE] Crazy special checker attacks have to work with all of that [WAIT]. Suddenly there's a whole lot of complex data getting passed around. Your code works fine for a while,
[ADVANCE]But then one day pieces start changing color inexplicably. You've somehow screwed up your message passing, and it's going to take days to track it down.

Goals

- DRY - Don't Repeat Yourself
- Eliminate boilerplate to reduce bugs
- No more hand-coded serialize/deserialize
- Spend more time on the game, not the protocol
- Build a helpful robot that writes our code for us



[ADVANCE]I like code that is bug free. Even more, I like code that doesn't let me write bugs in the first place. DRY, or don't-repeat-yourself, is an oft-repeated acronym among programmers.

[ADVANCE]The idea is that if you have to write the same code more than once, you will create more errors. Reducing the amount of code you have to write to accomplish a task also reduces the number of bugs you have to find later. Any time you find yourself writing the same code, or similar code, over and over again, you should stop and figure out how to write that code just once and be done with it. I'm going to outline a couple of things we've done to automate the repetitive parts of network programming.

[ADVANCE]One of the problems that JAM was originally created to solve was errors due to hand-serialization and de-serialization of data. This is bad, don't do it. JAM isn't the only way to automatically handle this problem, and I'll give you some open-source alternatives to doing it yourself at the end of this talk, but no matter what, don't do it by hand every time! Constructing messages is only part of the solution that JAM gives us. We also need to be able to get our messages from one place to another. I will also show you how JAM manages our connections for us.

[ADVANCE]Protocols and routing are mostly boring! We're here to write games, not tell stupid computers how to tell other stupid computers things. Figuring out what information needs to get from one place to another is our job as programmers. We should be able to tell our computers the bare minimum that they need to know in order to do that and let them figure out the hard parts. Computers are really good at doing boring stuff, put them to work for you.

[ADVANCE]JAM gives us a helpful robot that writes code for us. Using JAM, we describe a protocol using familiar syntax. From that description, we generate code in C++ and other languages to handle serialization. Our message handlers are well-defined and adding a new message is as easy as declaring a new structure. Connecting to remote services is simple and is managed for us as much or as little as we want. JAM leaves us free to concentrate on the parts of our games that matter, and does so with very little cost in bandwidth or processing time.

Goal: Human readable code

```
struct CheckerCaptured {
    CheckerID id;
    CheckerID capturedBy;
    u8 jumpType;
};

void Capture(CheckerID id, CheckerID by, JUMP_TYPE jumpType)
{
    CheckerCaptured msg;
    msg.id = id;
    msg.capturedBy = by;
    msg.jumpType = jumpType;
    Send(&msg);
}
```

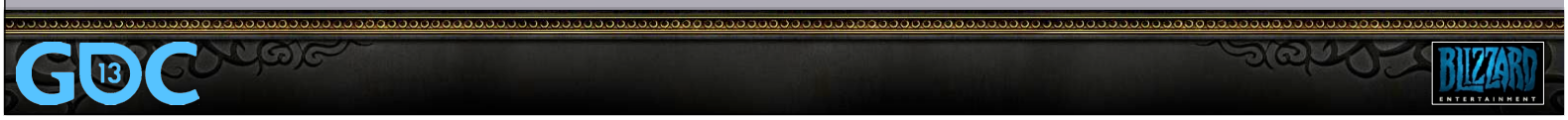
This is how we really want to write our protocols, right?
[ADVANCE]There's a structure definition for our message,
[ADVANCE]a simple declaration,
[ADVANCE]we fill some data in,
[ADVANCE]and call send.
[ADVANCE] All of our communication should be this easy. When we need to send another program a message saying a checker has been captured or a monster has been killed, the message should be a structure named CheckerCaptured or MonsterKilled, with a field saying what checker or monster it was, who captured it or killed it, and anything else the receiver might need to know about it. There should be a simple structure per event that needs to be sent, and sending one of these messages should be possible using nothing but standard C++ code. When we need another kind of message, we write another structure definition and we're done with it. The receiving code should be a function that gets a pointer to one of these whenever another program decides to send us one. No messy serialization details to deal with, no chance anyone's going to parse it incorrectly. Computers are good at figuring this stuff out, we should let them.

Implementation Details

Ok, that's it for theory. Let's get into the meat of this talk, how we actually do this stuff.

Development Cycle

- Describe the protocol
- Generate serialization and dispatch
- Send messages
- Receive messages
- Configure routing info



There are five steps to creating and using JAM in an application. The first is to describe a protocol. Let's see how that looks.

1-to-1 mapping of .jam messages to C++ classes

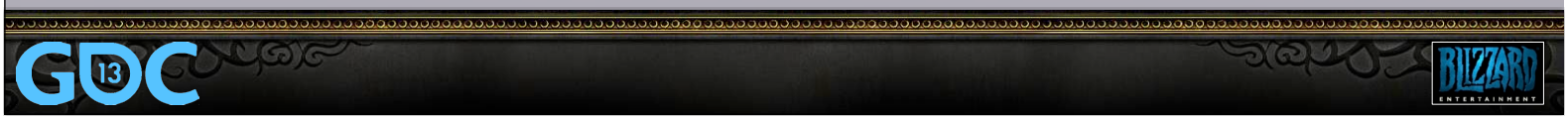
```
// From Checkers.jam
message CheckerCaptureCredit {
    CheckerID capturedCheckerID;
    CheckerID capturedBy;
    u8 jumpType;
};
```

```
// 100% Generated code in JamCheckers.cpp
class CheckerCaptureCredit : public JamMessage {
public:
    // Message decoders
    BOOL Get(BinaryDecoder &decoder);
    BOOL Get(JSONDecoder &decoder);
    // Message encoders
    BOOL Put(BinaryEncoder &encoder) const;
    BOOL Put(JSONEncoder &encoder) const;
    /**** DATA START ****/
    CheckerID capturedCheckerID;
    CheckerID capturedBy;
    u8 jumpType;
    /**** DATA STOP  ****/
    // Lots more stuff...
};
```

This is a sample JAM message definition. It looks a lot like a plain old C struct, on purpose. A JAM protocol is really just a bunch of these message definitions in a .jam file. All of the data members that we put in JAM message definitions have a direct correspondence to members in the C++ structures we generate. That may sound obvious, but it's important. If you look at a .jam file, you know immediately what the structures are going to be named, what the fields in those structures are, and how to write message senders and handlers for them. Also from this file you know what the protocol is going to be named, which will be important later when setting up our program to send and receive these messages. [ADVANCE] If you do need to see the actual code, we try to generate code you'd be mostly happy to have code reviewed by your peers. We can see here that our message definition has turned into a class with the same name that inherits JamMessage. JamMessage is an interface class that implements virtual methods for serialization and other things we might like to do with a generic message.

Development Cycle

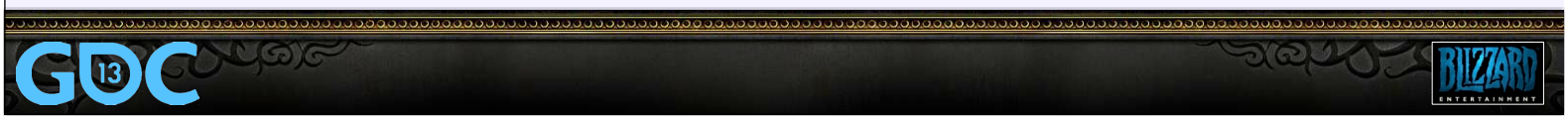
- Describe the protocol
- **Generate serialization and dispatch**
- Send messages
- Receive messages
- Configure routing info



The next step after describing a protocol is to run that description through a code generator that generates a bunch of serialization and dispatch code.

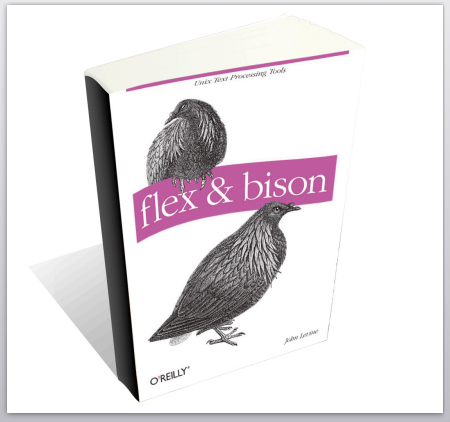
Auto-generated serialization code

```
//NOTICE: This is generated code.  DO NOT EDIT!
BOOL CheckerCaptureCredit::Put(BinaryEncoder &_encoder) const
{
    _encoder.BeginMessage(CODE, NAME);
    _encoder.Put("capturedCheckerID", capturedCheckerID);
    _encoder.Put("capturedBy", capturedBy);
    _encoder.Put("jumpType", jumpType);
    _encoder.EndMessage(CODE, NAME);
    return TRUE;
}
```



All that ugly serialization code we saw before is now generated for us in the output from the jam parser. Given a BinaryEncoder object, which is basically a stream, any JAM message can serialize itself to that stream. Then later, using a BinaryDecoder object with that stream as input, a JAM protocol can create and dispatch an instance of the C++ message classes we saw previously. With all those capabilities, we now have enough to implement a simple network protocol. That level of payload serialization is as low level as I'm going to get in this talk. The assumption is that you have encoder and decoder objects that can deal with simple data, and a transport layer underneath that to send it over the wire. JAM generates code to serialize a message using an encoder, an encoder creates a raw stream of bytes, and a transport layer gets those bytes where JAM tells them to go. Later I'll talk about how we negotiate which protocols to use over a given connection, and how we route messages, but the actual transport layer and encoding are out of the scope of this talk.

Flex and Bison make writing parsers easy



Flex & Bison - parser generators

Other tools

- ANTLR
- GOLD
- PLY (Python Lex & Yacc)
- Boost.Spirit

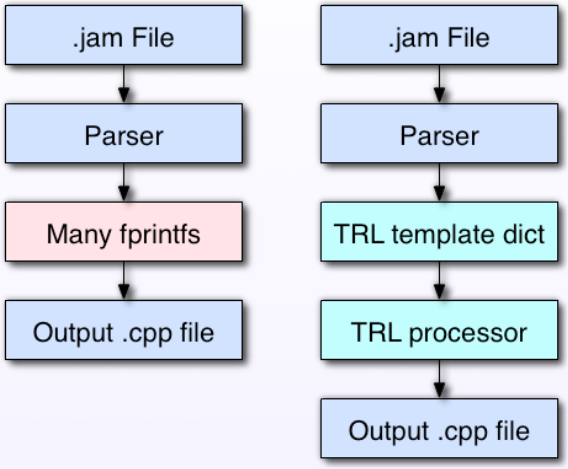
So far we've seen a JAM message and some of the code generated for that message, but if you haven't seen something like this before, you might be asking, "Hey, Joe, how do you get from a .jam file to C++ code you can actually use?" I'm glad someone asked! A really long time ago, some really smart people created some programs to help parse structured languages. Lex and Yacc, or their GNU replacements Flex and Bison, are possibly the best known of these. More recently there's also Antlr, which you might look into if you are looking at setting up a system like JAM on your own. I use Flex and Bison because they work really well and I've used them many times before. When I started working on WoW I was young and dumb and Google barely existed. I already knew how to use Flex and Bison, so I used them again. All of the tools listed offer way more power than you're likely to use for this particular task. I don't think you can actually go wrong with any of them in this case. If you're new to this, ANTLR and GOLD both include GUIs that may make it easier to get started, or if you decide to go with Flex and Bison, the O'Reilly book is a great way to learn them.

JAM File syntax is described to Bison

```
jamstructs      : jamstructs jamstruct
                | jamstruct
                ;

jamstruct       : structtype TIDENTIFIER messagename '{' { jam_start_s
                | structtype TIDENTIFIER messagename '{' '}' ';' { jam
                | jamcomment { /*printf("Comment\n");*/ }
                ;

structtype      : TMESSAGE { $$.type = GMESSAGE; $$.qualifiers = 0; }
                | TOBJECTMESSAGE { $$.type = GMESSAGE; $$.qualifiers =
                | TSTRUCT { $$.type = GSTRUCT; $$.qualifiers = 0;}
                ;
```


This is hard to adequately describe in the time I have available, but the really quick description is that Lex and Yacc together let us very clearly describe the syntax of a .jam file in such a way that we can load the contents of a file into a format that we can work with easily in code. When we load a .jam file, we get a syntax tree representing the metadata of all the messages in that file. We can iterate the fields of a message, get their names and types as strings and enum values, and retrieve other attributes of various things that are present in the .jam file. We've essentially created half of a compiler – that is, the parser – for half of a programming language – the data definitions. Yacc stands for Yet Another Compiler Compiler, and this is exactly the kind of thing it was intended for. Yacc and jamgen, the program that the parser gets compiled into, have a lot in common. They both exist to make up for shortcomings in other programming languages that were causing their creators pain. Both take simple inputs and produce output that would be tedious or difficult to produce by hand.

From .jam to .cpp



This shows the basic flow to get from a .jam file to a .cpp file. First we load a representation of a .jam file into memory using Bison. We still need to turn it into C++. How do we do that? Back when WoW shipped, jamgen just had a bunch of code that walked all the structures in the parse tree and fprintf'ed it to files. Oh sure, there were fancy macros and helpers and so on, but that's really all it was. This worked up to a point, but as things got more complex, it got harder to change the output code. And then we started writing more tools for other problems that needed to do similar things. That's when TRL was born. TRL is the Template Replacement Language. Jamgen loads a .jam file, puts all the data into a dictionary, and hands it to TRL.

TRL Turns .jam into C++

```
{@ define OutputMessage(msg, encoders, decoders) @}
    //
    // NOTICE: This is generated code. DO NOT EDIT!
    //
    class {{ msg.structName }} : public JamMessage {
    public:
        static u32 CRC;
        static u16 CODE;
        static cchar *NAME;

        // No argument constructor:
        {{ msg.structName }}() {

            {@ foreach f in msg.fields @}
                {@ if f.hasDefault @}
                    {{ f.name }} = {{ f.defValue }};
                {@ end if @}
            {@ end foreach @}

        }
```

TRL to generate a message class definition

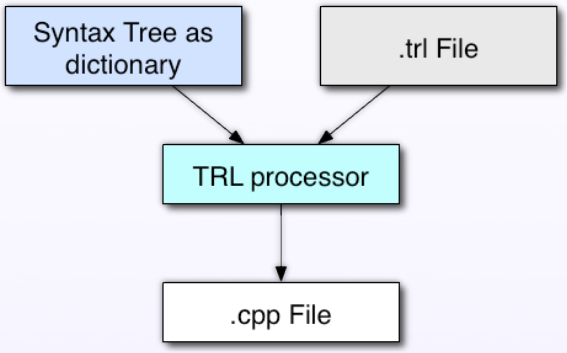
See Also

- CTemplate
- ngTemplate
- Django (HTML focused)
- Jinja (Python)

This is what TRL's input looks like. In this case, it's a template to generate C++ code. Every JAM message we put in a .jam file gets run through this template to produce the generated message class. It's a little bit ugly, but it's a whole lot better than a bunch of printfs.

[ADVANCE]If you're interested in this sort of thing, TRL was originally inspired by Django's HTML template language. For a ready to use C++ library, Google's CTemplate is similar, as is the pure-C ngTemplate. When I gave this talk at Blizzard, someone mentioned to me that they're using Jinja, spelled like Ninja, another standalone python templating library.

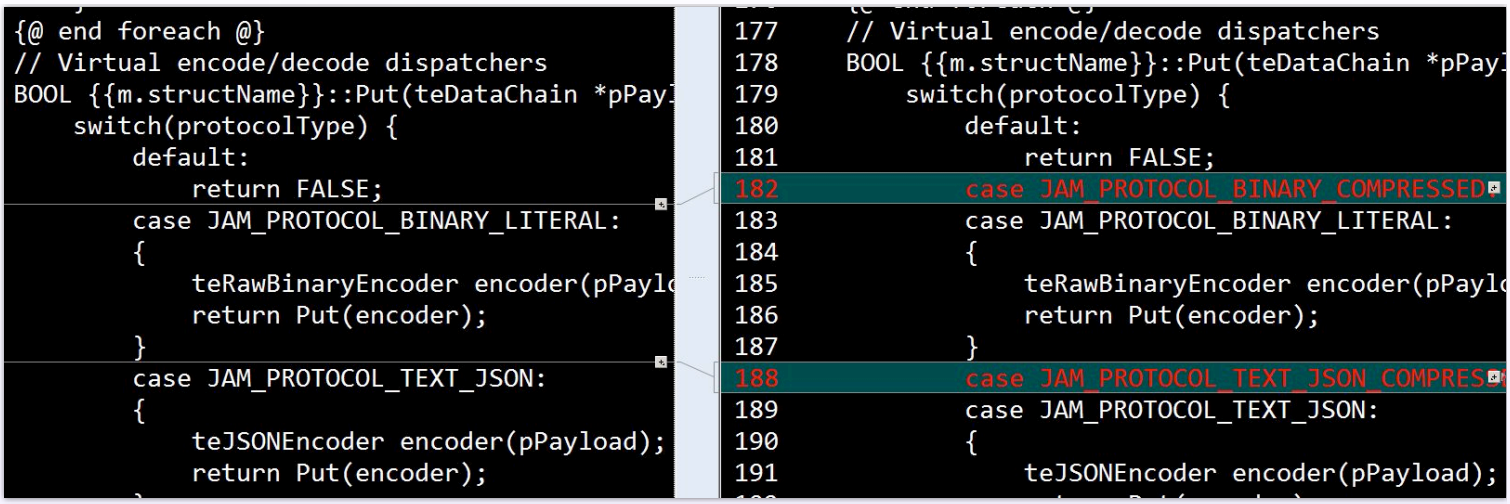
Fill out a dictionary and feed it to TRL




```
{@ foreach f in msg.fields @}
    {@ if f.hasDefault @}
        {{ f.name }} = {{ f.defValue }};
    {@ end if @}
{@ end foreach @}
```


Now that we've got our syntax tree and a TRL file that looks vaguely like the code we want to produce, what next? It's easy! Walk the syntax tree and fill out a string based dictionary containing all the relevant data. Hand that dictionary and a TRL filename to the TRL library, and it will spit out our output file. All templating engines work that way, CTemplate or something like it should work just fine for this. The code that builds the dictionary is simple. It has a linked list of messages, which it iterates, and for each message adds a list node to the "messages" key in our dictionary. In each node, it adds the name of the message and some other metadata. Then it adds a list to the message which will contain all the fields of the message.
[ADVANCE]On the previous slide, we saw the TRL code that iterated the fields of the message and added default values to the generated constructor for each field that has a default value. That's it up there, you can see that besides [ADVANCE]the simple replacements in the middle, TRL also has [ADVANCE]some simple control structures. In fact, that snippet contains about three quarters of all the language features in TRL. The only thing it's really missing is function calls.

Global Feature addition using TRL



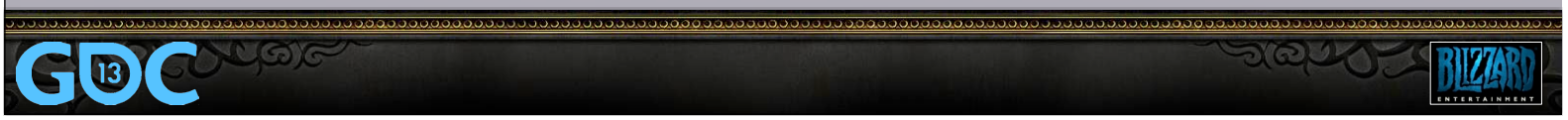
```
{@ end foreach @}
// Virtual encode/decode dispatchers
BOOL {{m.structName}}::Put(teDataChain *pPay
    switch(protocolType) {
        default:
            return FALSE;
        case JAM_PROTOCOL_BINARY_LITERAL:
        {
            teRawBinaryEncoder encoder(pPaylo
            return Put(encoder);
        }
        case JAM_PROTOCOL_TEXT_JSON:
        {
            teJSONEncoder encoder(pPayload);
            return Put(encoder);
```

```
177     // Virtual encode/decode dispatchers
178     BOOL {{m.structName}}::Put(teDataChain *pPay
179         switch(protocolType) {
180             default:
181                 return FALSE;
182             case JAM_PROTOCOL_BINARY_COMPRESSED
183             case JAM_PROTOCOL_BINARY_LITERAL:
184             {
185                 teRawBinaryEncoder encoder(pPaylo
186                 return Put(encoder);
187             }
188             case JAM_PROTOCOL_TEXT_JSON_COMPRES
189             case JAM_PROTOCOL_TEXT_JSON:
190             {
191                 teJSONEncoder encoder(pPayload);
```

When we need to make a change to all messages, it's easy now. Say we had originally decided a Checker ID was a byte, because 256 checkers should be enough for anyone. Since Checker IDs are used in virtually every message in our checkers MMO, we built CheckerID into jamgen as an actual type that's really just a typedef to uint8. Later though we realized that 256 is not nearly enough checkers for an MMO and we need to change it to a u16. If we were hand serializing all our messages, this could be a disaster. But since we're not, we can just change a few things in jamgen or likely even just in our trl and suddenly we've fixed all of our protocols. This is a bit of a contrived example since we don't generally add special types to jamgen itself, but the idea that we can make changes to many protocols at once holds. On the slide you can see an actual change where we added the ability to compress JAM messages. There was a little bit of extra handling needed in our serializers, which we got automatically for all messages by writing the code once in our TRL file. This two line diff would have replicated this change across dozens of protocols and hundreds of messages.

Development Cycle

- Describe the protocol
- Generate serialization and dispatch
- Send messages
- Receive messages
- Configure routing info



Now that we've got a protocol described, and have generated code for that protocol, we're ready to write application code that sends the messages from that protocol.

Create a message, fill in data, call send

```
void Checker::OnCaptured(CheckerID capturedBy, JUMP_TYPE how)
{
    CheckerCapturedCredit msg;
    msg.capturedCheckerID = GetID();
    msg.capturedBy = capturedBy;
    msg.jumpType = how;
    JamID destination = GetRouter()->GetCreditManagerID();
    GetRouter()->Send(destination, &msg);
}
```

This is how a JAM message gets sent.
[ADVANCE]A simple instance of the CheckerCapturedCredit message is declared.
[ADVANCE]Next we put this checker, the one that just got jumped, into the capturedCheckerID field.
[ADVANCE]Since we know who jumped this checker, we put that in the message too. The last bit of data is what type of jump it was. You know, like a regular jump, a double jump, a critical jump or special ability jump.
[ADVANCE]The destination creditManager is a JamID, which is really just a 128-bit address which can define either a service or an object. Later, I'll show alternate ways of sending messages to service **TYPES** rather than specific services, and how we send messages directly to objects too. For the most part, JamIDs are opaque to application code though. You can compare them and copy them, but the contents aren't important.

[ADVANCE]
Each individual JAM message is just a structure. When it's time to create a message to send, all you need to do is delcare an instance of one and fill out the data. It's simple C++, practically simple C, and there's nothing hidden going on in this code. Though this message, like most messages to be sent, is declared as a local variable on the stack, there's no reason we can't allocate it with new or make a pool of messages or any other allocation pattern we want to use. There are a few high volume uses of JAM where we do things slightly differently just to avoid the small construction and destruction overhead. But 99% of the places that send JAM messages do it just like this.

Structs and arrays in messages

```
message GroupUpdate
{
    GroupID group;
    array<.CheckerID> checkers;
};
```

```
    /*** DATA START ***/
    GroupID group;
    vector<CheckerID> checkers;
    /*** DATA STOP ***/
```

```
void GroupService::SendUpdate(GroupID id)
{
    GroupUpdate msg;
    msg.group = id;
    msg.checkers.resize(MAX_GROUP_SIZE);
    // ...
}
```

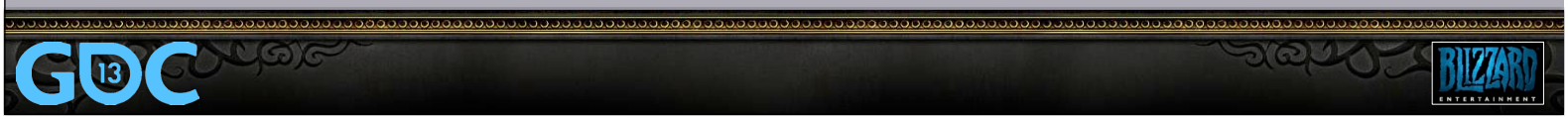
So far I've only shown simple types as data in JAM messages, but JAM allows us to use dynamically sized arrays in messages, and also structs and arrays of structs. Originally, "struct" support was just the ability to use any C++ class that implemented serialization methods (horrible, awful, hand-written serialization, have you figured out how much I hate it?). That method still works, for protocols where only C++ output matters, but we have a better way now. We have a reflection system for generic non-message data called "structgen". The structs it outputs are also used by things other than just JAM, like our database and asset layers, which is why it's not just part of jamgen too. Structgen is built using TRL just like Jamgen. The structures that structgen outputs, and arrays of those structures, can be embedded in JAM messages just like plain old data.

[ADVANCE]On the upper left, you can see that I defined a message named GroupUpdate that contains a group ID and an array of checker IDs.
[ADVANCE]In the upper right is the data section of the generated JamMessage definition containing a standard vector of checker IDs.
[ADVANCE]Finally on the bottom there's some code that fills in that message, resizing the vector first.

We don't actually use STL in our code, generated or otherwise. We have our own container classes, but the reasons for that are way outside the scope of this talk. I've tried to use STL for all the examples in this talk just so it'll be clear to everyone what's going on. If you catch any non-STL containers in here, give me a hard time later.

Definitions

- Message - serialized structure defined in a .jam file
- Protocol - a collection of messages
- Service - a module of code that implements message handlers for one or more protocols
- Program - can be composed of multiple services



Before I go on, let me make sure I've defined some terms I'm using a lot.

- [ADVANCE]Message - serialized structure defined in a .jam file
- [ADVANCE] Protocol - a collection of messages
- [ADVANCE] Service - a module of code that implements message handlers for one or more protocols
- [ADVANCE] Program - can be composed of multiple services

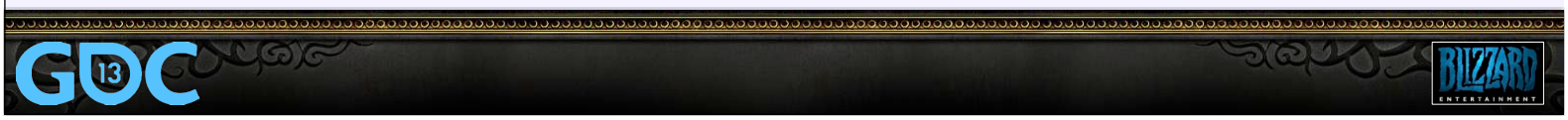
Message Destinations

```
void MatchService::CreateBoard(u64 width, u64 height) {
    BoardID = GenerateBoard();
    // Send to a known, connected, service
    m_pServer->Send(m_boardServerID, &msg);
}

void MatchService::GameOver(u32 gameID, u64 winnerID) {
    msg.gameID = gameID;
    msg.winner = winnerID();
    // Send to a service type, non-specified ID
    m_pServer->Broadcast(JAM_SERVER_TYPE_TRACKER, &msg);
}

void Checker::HealChecker(CheckerID toHeal, u32 amount) {
    CheckerHeal msg;
    msg.healedBy = GetID();
    msg.amount = amount;
    // Send a message to a specific object
    m_pServer->Send(toHeal, &msg);
}
```

JAM provides us with several different ways to send a message.
[ADVANCE] Sometimes we know the JamID of a specific service we are connected to. We can call a send method with that JamID and a message pointer and it will get there.
[ADVANCE]Sometimes all we know is a type of service and we want to send a message to any service of that type that we're connected to. We can do that too.
[ADVANCE]We can also broadcast a message to all connected services of a specified type. In some cases, we have a lot of different objects, checkers for example, that all live within a larger service.
[ADVANCE]We can also send messages to the individual objects. If you're looking closely, you'll notice the third example is sending to a CheckerID rather than a JamID. They are related types, they are both 128-bit IDs, and services that use objects as destinations can use a CheckerID as a destination just as easily as a JamID as long as the receiver provides a way to resolve a CheckerID to a Checker object in its dispatcher.

Message routing by type

```
MatchmakerAddPlayer addMsg;
addMsg.player = GetPlayerID();
addMsg.rank = GetRank();

// No JamID needed, send to any Matchmaker
// May be queued until a Matchmaker is available
m_pService->Send(JAM_SERVER_MATCHMAKER, &addMsg);
```



Until the last couple of years, most of our messages were sent to specific JamIDs – either services or objects. JAM provides Link up and link down notifications from which we can retrieve JamID addresses. As we started implementing more and more services and tools using JAM, we noticed that there was a similar pattern to many of them where they would watch for a service of a specific type, store off the JamID whenever they saw the first one, use it until they lost connection to that service, then find another service of that type and start over. That code was also really easy to screw up, and we kept finding subtly different ways to do it wrong. So we added a class that does two things for us. First it tracks service connections of types we've tried to send a message to, so that we can just tell it to send a message to a service type rather than a specific instance of one. Second, it has a message queue that gets used only if there are no valid services of the specified type. That way, we can now "send" messages even before we have a valid connection, or after we've lost one and before a replacement connection has been found. This is a poor-man's message queue in the RabbitMQ/ActiveMQ sense, and doing JAM over AMQP is something we've been looking at lately too, though we don't do it yet.

Send a message and expect a response

```
MatchmakerAddPlayer addMsg;
addMsg.player = GetPlayerID();
addMsg.level = GetLevel();

// Send to any Matchmaker, PlayerAddedHandler
// will be called with response when complete
m_pService->SendRegistered<PlayerAdded>(
    JAM_SERVER_MATCHMAKER, &addMsg
);
```

Many networking libraries implement an RPC mechanism. JAM doesn't exactly do this, but it can pretend it does. We have a notion of registered messages, and can associate specific response messages with a request we sent. In this example, we see code that is sending an AddPlayer request to a Matchmaker. [ADVANCE]The template argument indicates that we are expecting a PlayerAdded message as a response. Behind the scenes, what happens here is JAM inserts an integer ID sequence into the message header, and then the Matchmaker sends a reply back with that same sequence. Though the main reason for the template parameter is to make the flow clear when reading the code, we also check when we get the response that the type is actually correct. We can do this without any extra work because the message is a generated class, and one of the generated methods is a static GetMessageID method.

Send a message to an object

```
void CheckerGroup::ChangeBoards(u32 newBoard)
{
    CheckerChangeBoard msg;
    msg.boardID = newBoard;
    for(int i = 0; i < m_checkers.size(); i++) {
        m_pServer->Send(m_checkers[i]->GetID(), &msg);
    }
}
```

Although a lot of our JAM based code is services and tools talking to other services and tools, we are, believe it or not, making a game. In this game, we have a bunch of objects – checkers – that need to talk to each other. Sometimes our checkers live all on one server, sometimes they're spread across many servers. When we want to send a message from one checker to another, we could of course figure out what server the other checker is on, send that server a message that contained an ID for the recipient player, and that server could then call a method on that player with the message we sent it. That's a lot of work! We'd rather do what this slide shows. It has an array of checker pointers, and sends a copy of the same message directly to each of those checkers.

Each object is owned by one server

```
class Checker {
    //...
    CheckerID m_id;
    JamID m_serverID;

    JamID GetServer() {
        return m_serverID;
    }

    CheckerID GetID() {
        return m_id;
    }
    //...
};
```

In our checkers MMO, each checker is owned by exactly one server. But checkers on different servers can still interact with each other. To accomplish this, read-only copies of checkers are mirrored to other nearby servers. Whenever one checker wants to interact with another, it can only do so by sending that other checker a message. When we receive a mirrored copy of a checker, we record what server actually owns it,
[ADVANCE]as you can see on the slide in the m_serverID member. Even for checkers owned by the local server, we just put the local server's ID into the m_serverID field. One important aspect of JAM, especially for messages sent to objects, is that loopback messages are allowed. In many cases involving multiple services of the same type, our code will be cleaner if we write it to always assume message passing is required. Because we can send a message to our own server's JamID, those cases will work transparently whether the object is remote or local.

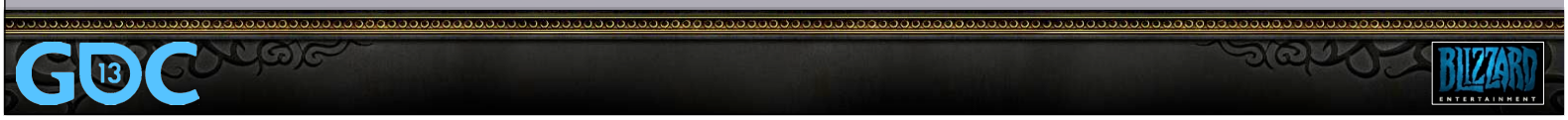
How messages get routed

```
void BoardServer::Send(Checker *pChecker, JamMessage *pMessage)
{
    m_pJamServer->Send(pChecker->GetServer(),
                       pChecker->GetID(),
                       pMessage);
}
```


I've shown a few things calling server send methods. In programs that deal with service level messages only, there are pre-defined Send methods built into JAM. Services that deal with object level messages typically implement one like this that determines from object-specific data what server the object being sent to lives on, and then calls a built-in JAM method that also passes an object ID on to the receiving end. This is typically written once per service, and if there are multiple protocols handled by that service, they can all share this method.

Development Cycle

- Describe the protocol
- Generate serialization and dispatch
- Send messages
- Receive messages
- Configure routing info



So we're sending messages, but that doesn't do us much good if there's nothing that knows how to work with those messages. Let's see how we write message handlers now.

On receipt, look up and dispatch

```
// static callback registered with JAM by protocol ID
// called for each incoming message
void BoardServer::CheckerDispatch(JamLink &link, JamMessage *pMessage)
{
    CheckerID destID = pMessage->GetDestination();
    Checker *pChecker = GetCheckerObject(destID);
    pChecker->QueueMessage(pMessage);
    switch(pMessage->GetProtocolCRC()) {
        case JAMCheckerProtocol_CRC:
            JamCheckerProtocol::Dispatch<Checker>(pMessage, pChecker);
    }
}
```

On the receiving end, we get a link and a message pointer for each message that comes in on protocols that have been specifically registered for our service. The message object now contains the destination checker ID that the sender asked us to deliver this message to.
[ADVANCE] What that means is we can look up an object, seen here in the GetCheckerObject call,
[ADVANCE]then call a dispatch method that will in-turn call a method on that object with our message pointer. The JamCheckerProtocol::Dispatch method called in the switch here is a method that was generated from our .jam file.
[ADVANCE]To be clear, the code on the slide is code we only have to write for protocols that are to be dispatched to objects rather than services. JAM will include object IDs in the messages it sends, but needs the application to lookup objects to which messages can be dispatched. That is all this code does. We still didn't have to write anything here that is aware of specific message types, only protocols. And depending on the service, it might be possible to assume all messages are object messages and not even have to write a case per protocol. In reality, our main gameplay server actually routes messages to components based on the protocol, not just to base level objects.

JamLink

```
void BoardServer::CheckerDispatch(JamLink &link,
                                  JamMessage *pMessage)
{
```

You'll see some examples that pass a JamLink reference around, especially to message handlers and dispatchers. JamLink is an object representing a connection to a remote application. From it, we can tell what type of app we're talking to, what protocols it supports, and disconnect it if it's misbehaving somehow. It's a straightforward class, I just wanted to point it out so you know there's nothing magic going on when you see it. Most handlers only care about the message they get passed, but a few need more information about the sender than just the JamID, so we pass them a JamLink as a convenience. It's also possible to go from a JamID to a JamLink at any time, except that outside a handler, there's no guarantee the JamLink will still exist if the application at the other end has gone away. Inside a handler, the JamLink that corresponds to the sending application is always guaranteed to be valid.

Generated Dispatch methods

```
//NOTICE: This is generated code.  DO NOT EDIT!
template<typename HANDLER_T>
static JAM_RESULT Dispatch(JamMessage *pMessage,
                           HANDLER_T *pHandler) {
    switch(pMessage->GetCode()) {
    case JAM_MSG_CheckerHeal:
        result = pHandler->CheckerHealHandler(link,
            (CheckerHeal *)pMessage);
        break;
    // cases for rest of protocol's messages...
```

In the previous slides, we saw hand-written code that understood application specific objects and called a dispatch method. This is the generated dispatch method it was calling. You can see that it takes a templated type, we passed it a checker, a pointer to that type, and a message pointer.

[ADVANCE]The message contains a numeric code that can be mapped to a specific message handler. So we just call that method on the object we were passed. This is one of the big benefits of code generation. If we had to write this dispatch method by hand for every protocol we implemented, we would be very sad. CheckerHealHandler, and any other message handlers required for this protocol, are methods whose bodies we define by hand, but whose prototypes are generated for us. Let's see how that works.

Generated message handler prototypes

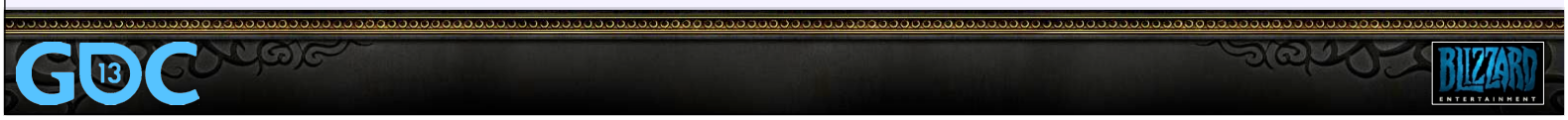
```
// A message handler prototype is auto-generated for each message
// in the protocol.  #include these declarations in the middle
// of your hand constructed class.
JAM_RESULT CheckerHealHandler(JamLink &link, CheckerHeal *msg);
JAM_RESULT CheckerDamageHandler(JamLink &link, CheckerDamage *msg);
JAM_RESULT CheckerPowerupHandler(JamLink &link, CheckerPowerup *msg);
JAM_RESULT CheckerKingHandler(JamLink &link, CheckerKing *msg);
```

#include this in the middle of a class

When we run our .jam files through jamgen, one of the output files is a header file that's really just a code snippet containing prototypes of message handlers. The general usage is to #include that header in the middle of a class in order to add message handlers for all messages in a protocol. I know that sounds a little bit odd. It is a little bit odd. In the past, we generated pure virtual interfaces instead. Which resulted in lots of multiple inheritance and a little bit of extra overhead in calling virutal methods. But either approach works, and it's mostly down to aesthetics which you chose. In either case, the generated dispatch method on the previous slide is calling the methods defined by this header. You could also implement something similar with a macro that expands to all of the required prototypes, but I find this method to be slightly cleaner.

Message handler methods

```
JAM_RESULT Checker::CheckerHealHandler(CheckerHeal *pMessage)
{
    m_health += pMessage->amount;
    LOG("Checker %d was healed for %d by checker %d",
        GetID(), pMessage->amount, pMessage->healedBy);
    return JAM_OK;
}
```



Finally we can actually handle our heal message! This is it, just a simple class method that takes a specific message pointer type, and can do what it wants to with the contents. When someone sends a CheckerHeal message to this checker, it will get dispatched to this method, which can then update the Checker's health, log some stuff, and be done. This is really the core of what JAM is all about, all the infrastructure I'm talking about today is mostly there so that we can implement these methods painlessly with a minimum of redundant code.

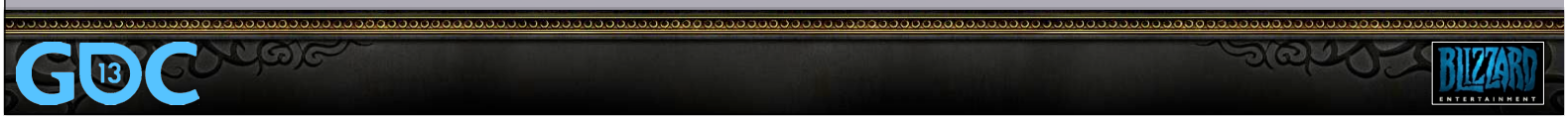
Send and Receive

```
void Checker::HealChecker(CheckerID toHeal, u32 amount) {
    CheckerHeal msg;
    msg.healedBy = GetID();
    msg.amount = amount;
    // Send a message to a specific object
    m_pServer->Send(toHeal, &msg);
}

JAM_RESULT Checker::CheckerHealHandler(CheckerHeal *pMessage)
{
    m_health += pMessage->amount;
    LOG("Checker %d was healed for %d by checker %d",
        GetID(), pMessage->amount, pMessage->healedBy);
    return JAM_OK;
}
```

To recap the last two sections, these two methods and one message definition are all the new application code we typically write to handle a new message type. There is some application code that knows how to route messages to objects, but we wrote that once a long time ago and forgot about it. When we add a new protocol, we might have to add a line or two to configure that protocol in our services. We don't have to add anything to it for new messages within a protocol. We write a function that sends a message, and we write a function that handles a message. Everything else is generated or written just once per protocol, not per message.

Development Cycle

- Describe the protocol
- Generate serialization and dispatch
- Send messages
- Receive messages
- Configure routing info

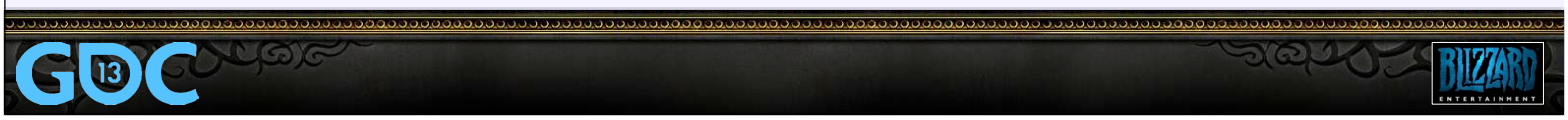


Finally we've reached the end of our cycle. We've written code to send and receive messages, but we haven't told the system how to get them from one place to another yet. That's pretty easy and I'll show you how we do it now.

Define services

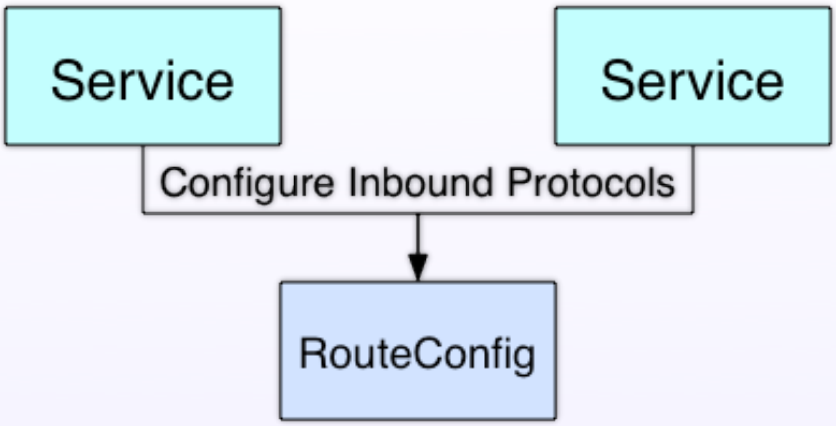
```
void Matchmaker::Configure(JamServer *pServer)
{
    JamRouteConfig &routeConfig = pServer->GetRouteConfig();
    routeConfig.ConfigureInbound<MatchmakerProtocol>(
        this, Matchmaker::DispatchMessage);
    routeConfig.ConfigureOutbound<MatchmakerResponseProtocol>();
}
```

Configure protocols the Matchmaker service sends and receives



JAM is based on services. In order to receive messages, we tell a JamRouteConfig object what protocols we want our JamService to handle. JamRouteConfig takes care of figuring out when and how to call our message handlers. In the simplest usage, all we have to do is call JamRouteConfig.ConfigureInbound with the protocol name we want to handle, and then implement the handlers for all the messages in that protocol. The prototypes for all those message handlers are already part of our class because of the generated file we #included a few slides ago. Though it's not obvious on this slide,
[ADVANCE]the first parameter we're passing to ConfigureInbound here can be an instance of any class that implements the handlers for the protocol we're configuring. Usually it's just an instance of the service, but it can be another class owned by the service if that makes more sense for the given application.

RouteConfig maintains a protocol to handler mapping



As we've seen, when we're setting up a program's protocols, we have a RouteConfig object, and we hand it protocol-to-handler mappings. For each mapping we give the RouteConfig, it creates an instance of an object that can dispatch messages on that protocol to the handler class type. By using templates here, we don't need virtual interfaces nor more generated code, though of course templates are just another way to generate code. Whenever a message is received, RouteConfig figures out what protocol it's a part of (more on that later), what handlers are available for that protocol, and how those handlers want those messages dispatched.

I'm saying handlers but the slide says services. Most of the time, a service registers itself as the handler for a given protocol. But that's not required, a different object can be set up as the handler if it helps make your service cleaner or otherwise makes sense.

Handlers have access to sender and other metadata about received messages

```
JAM_RESULT BoardServer::AddPlayerHandler(JamLink &link,
                                         AddPlayer *msg)
{
    LOG("Adding player %s from server %s",
        IDSTR(msg->playerID),
        link.Describe().c_str());
    // Do stuff
    return JAM_OK;
}
```

Besides the raw message data, each handler is passed a JamLink reference. From the JamLink, we can determine lots of things about the sender, such as what type of program it is (there's an enum that currently contains 45 different entries, each representing one application), a subtype that is defined on a per-application basis, and even some more specific stuff like map IDs (I mean board, checkers has boards, not maps!) when appropriate. Of course, the most important thing we get is the sender's unique JamID so that we can send them a response, track their request for later if it requires some asynchronous work, do authorization checks, or anything else we might want to do based on the sender.

Coarse and fine-grained queueing and

Race Condition

I mentioned that the simplest use of JamRouteConfig is to simply let it manage when and how we receive calls to our message handlers. That's sort of true, but due to its roots as something used only by high performance game servers, particularly multi-threaded ones, the simplest behavior doesn't actually do any thread synchronization for us, and it can and will call message handlers simultaneously from multiple threads (though it does guarantee messages from the same sender are never processed concurrently). It's really not what you want for the majority of applications, but luckily we have several other options available.

Receiving via Message Queue

```
void Matchmaker::Configure()
{
    // Messages received at any time are placed into a queue
    routeConfig.ConfigureInbound<MatchmakerProtocol>(
                this, &m_messageQueue);
}

void Matchmaker::Idle()
{
    // Queue is processed in one thread at a known time
    pServer->ProcessQueue(&m_messageQueue, this);
}
```

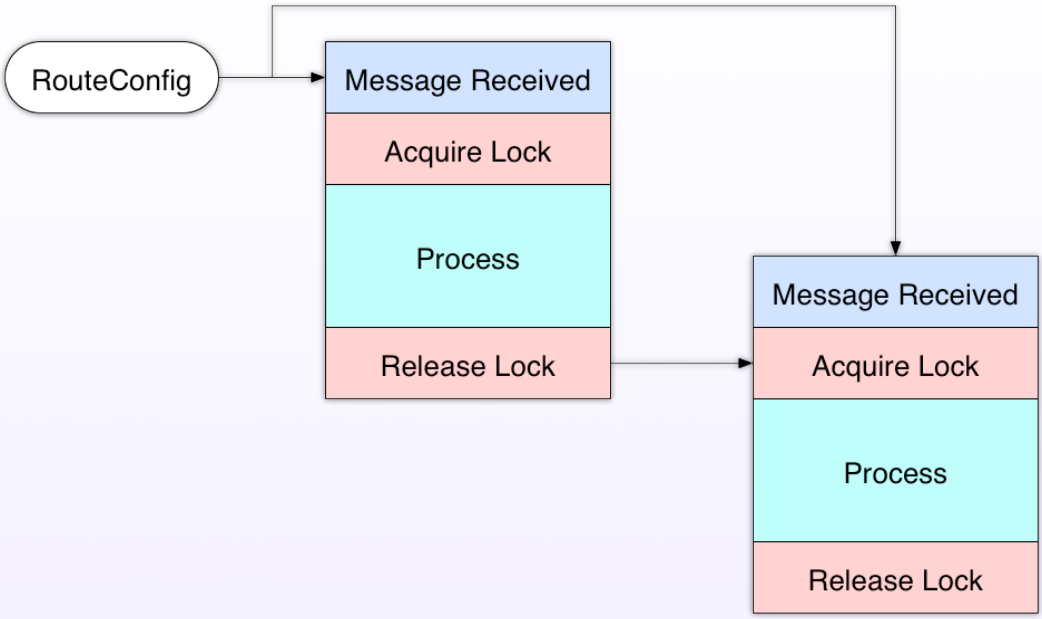
The easiest way to deal with receiving JAM messages is to give RouteConfig a message queue object for our service during configuration. [ADVANCE]Then we process that queue only at a known time on a single thread. This polled model is perfectly fine for the majority of our tools and servers and even for our game client. It is the preferred model anywhere where ease of use is more important than latency or throughput.

[ADVANCE] In this slide, we see some code that tells our route config to put messages for any of our supported protocols into a single queue, and

[ADVANCE] then in our Idle method we ask our JAM server to process that queue.

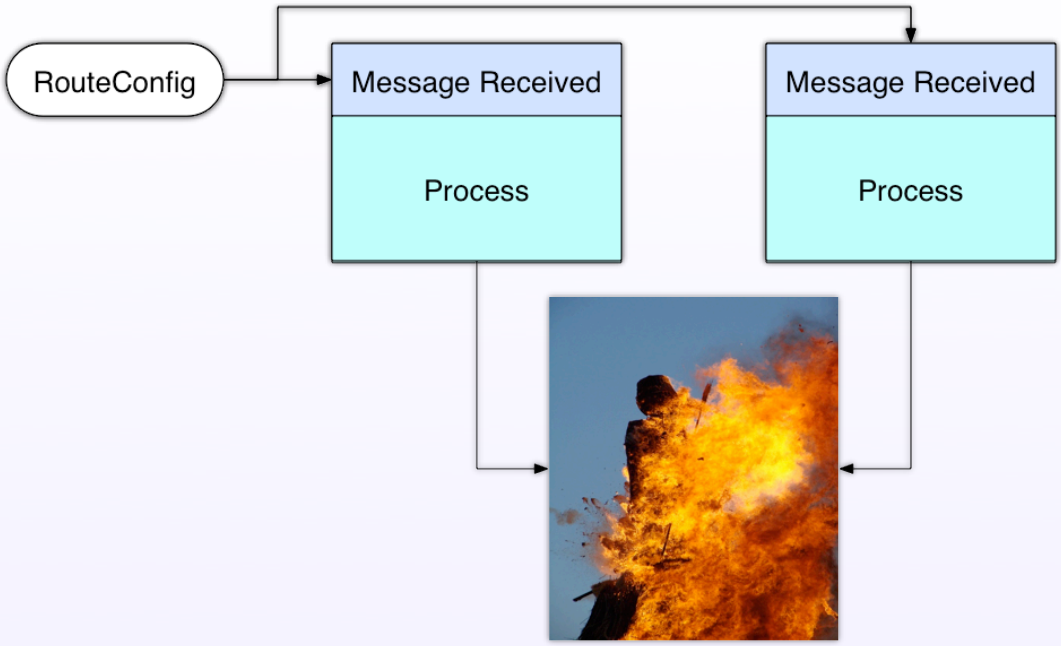
m_messageQueue in this case is a JamMessageQueue object, which is essentially just a linked list with internal locking that makes it safe for messages to be enqueued from any thread, and then swapped into a local list for processing later. That's what ProcessQueue does for us.

Global lock dispatching



Though I generally prefer to split servers into multiple processes rather than multiple threads, there are a few cases where having more threads makes more sense. There are two main strategies for these servers. One is to have a "global lock" in our service, which RouteConfig knows how to acquire and release, so that although our message handlers get called from multiple threads, only one of them is actually active at a time. While this makes it essentially single-threaded, it does have the advantage of not having to copy messages as much in some situations. For services that don't need a lot of concurrency during message handling, this method makes a lot of sense.

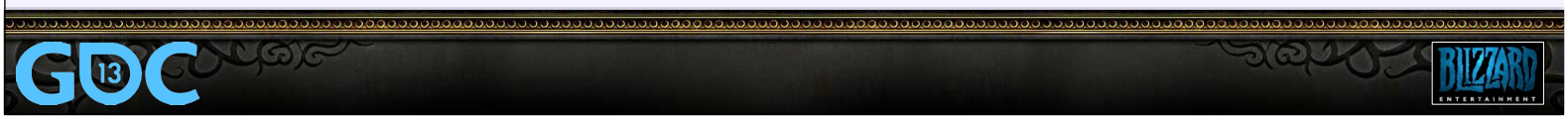
Raw concurrent handlers



It's also possible for services to manage locking themselves and not let RouteConfig do anything about it. Most of the time, this is the scary option you don't want to use. Though JAM will do the right thing as much as it can, this creates all the usual concurrent headaches in your service class, as messages are handled directly on a network thread with no guarantee you don't have other code running on another thread. How to deal with those kinds of concurrency issues is a whole 'nother talk, and I won't be going into it today. [ADVANCE] Other than to say this is what happens if you're not really careful with this method.

Lock Policies

```
class MatchmakerLockPolicy
{
    Matchmaker *m_owner;
    void Lock(JamMessage *msg, JamMessageQueue **ppQueue)
    {
        // Adding a player requires a write lock
        if(msg->GetCode() == JAM_MSG_MatchmakerAddPlayer) {
            m_owner->AcquireWriteLock();
        } else {
            m_owner->AcquireReadLock();
        }
    }
    void Unlock(JamMessage *msg) { /* Same logic, release lock */ }
}
```



Finally, services can actually create locking policies of their own, and hand those policies to RouteConfig, to handle locking on a per-message or per-protocol basis. This can be used to, for example, acquire a read-write lock in either read or write mode depending on the message being handled, or handle some messages immediately and defer others based on any application logic we want. The global lock strategy mentioned earlier is actually just an implementation of a lock policy that acquires and releases a single mutex for any incoming message. Since we pass this lock policy into the system as a template parameter, it doesn't need a base class, [ADVANCE]it just needs Lock and Unlock methods with the proper signatures.

Incoming messages are refcounted

- Message passed to handler is a refcounted object
- Possible to retain a message pointer until later
- Smart pointers are available
- Messages contain no pointers to any other objects
- No circular references are possible

A few last things on receiving messages.
[ADVANCE]The incoming message is actually a refcounted object.
[ADVANCE]If a message gets to our handler but we need to hang onto it for later because we're not ready to process it, or we need to do some I/O and then use something out of it, we are completely free to do that, as long as we retain a reference to it.
[ADVANCE]There are smart pointer classes for dealing with this, so we don't usually have problems with leaked messages,
[ADVANCE]Messages themselves don't contain references to anything,
[ADVANCE]Which means no circular references are possible either. [DO NOT ADVANCE]

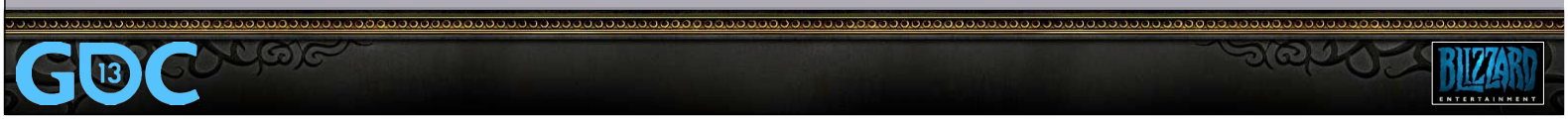
Though most services handle messages immediately and don't retain them, this flexibility is very handy for the few services that need it. In several cases, it lets us avoid reimplementing our own queueing systems by simply retaining a list of pending messages which we weren't able to process synchronously.

CPU And Bandwidth Efficiency

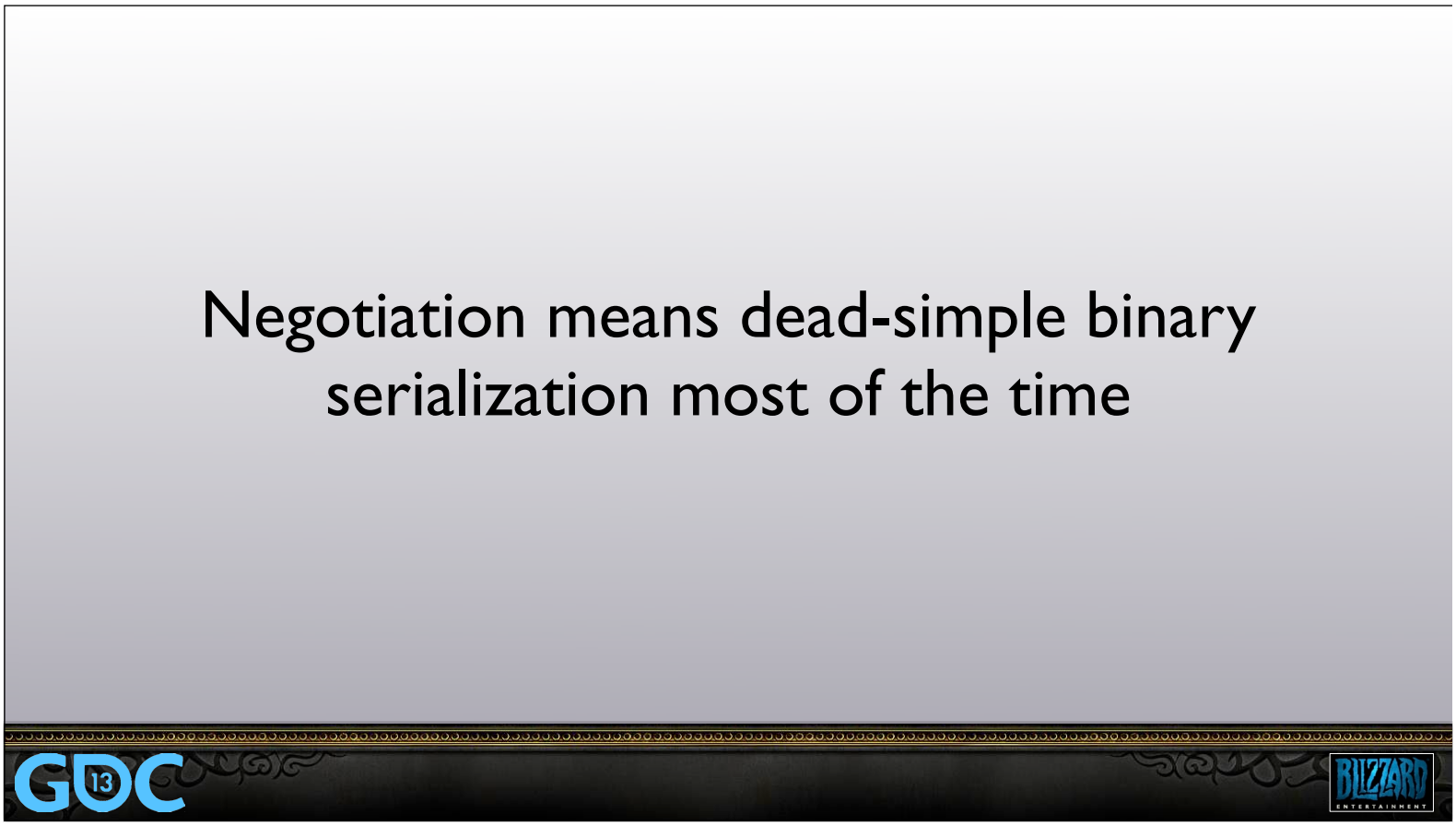
Since we're making a game and every cycle counts, we're always very concerned with how efficient our code is. Since we're also making MMO checkers and planning on hosting the servers ourselves, bandwidth efficiency is also important. I'm going to show you how JAM achieves these goals now.

JAM is either efficient or backwards compatible

2004 - Assumed binary compatibility



If you've dealt with the problems JAM attempts to solve in the last five years, you've probably run across things like Protocol Buffers, Thrift, or Avro. All of those and JAM are in a similar space. JAM's main differentiating feature is speed. JAM was originally used solely between game servers on World of Warcraft, and some of the scenarios it was used in were frighteningly high bandwidth (but not over the internet, all within a data center). Also, it was always safe (ish) to assume that anything you connected to was using the exact same protocol since servers for any given build were all deployed together. So back then, JAM strove only for high performance given pure binary compatibility. What it sacrificed was inter-version compatibility. There wasn't any. Once we started using JAM for tools and other things other than just game servers and clients, however, we quickly started having problems with having to synchronize our deployments of everything. I'm going to show you how we addressed that.



The first step to fixing incompatibilities between versions was to actually detect versions. JamGen today has the ability to create a CRC of each protocol it generates. If a protocol has all the same messages in the same order (we use numeric message IDs determined by the order they appear in their .jam file) and they all have the same fields, then the CRC comes out the same and two services using that protocol know their versions of it match exactly. They know this because the very first thing we send to the other end of a connection is a list of protocols we know how to send and receive and their CRCs. In this case, JAM knows it can use good old fashioned, fast, efficient, pure binary serialization. This is the case we expect to always hit for game servers.

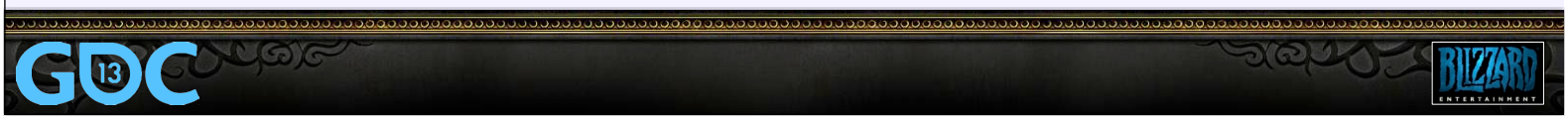
In some cases, can just memcpy it onto the wire

```
// This message could easily be memcpy'ed onto the wire
class CreateChecker : public JamMessage {
    /**** DATA START ****/
    u32 checkerType;
    u32 owner;
    /**** DATA STOP ****/
    // Code...
};
```


Because the wire format is so close to the structure layout, there are actually cases where we can simply memcpy a message onto the wire. When I say “we” in this case, unlike most of the rest of this talk, I actually mean World of Warcraft and not World of Checkers. I haven’t really mentioned it yet but there are actually two implementations of JAM at Blizzard. They share a common ancestor, but not a source repository. This is a case where WoW has added functionality we haven’t picked up yet. If jamgen can detect structure layouts that matches the wire format, and protocol negotiation can detect that our endianness matches, then we can simply copy messages and not even bother with “serialization”.

Generated code means easy optimizations

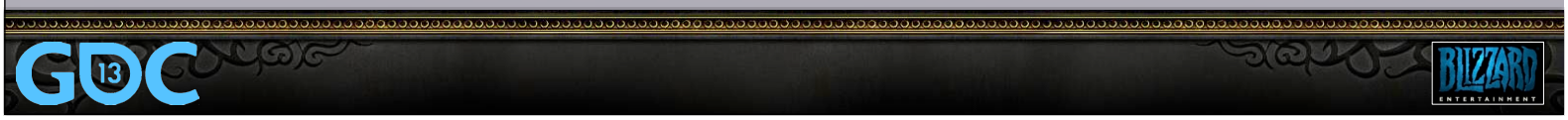
```
_encoder.Put("capturedCheckerID", capturedCheckerID);
_encoder.Put("capturedBy", capturedBy);
_encoder.Put("jumpType", jumpType);
```



Because we generate all of our serialization code, we can immediately take advantage of new optimizations across many different protocols all at once. This is true for both bandwidth and CPU optimizations. For example, [ADVANCE] you might have seen a few slides back that our binary encoder is doing a function call per field and passing a completely useless field name in. If we get smarter, we can easily generate serializers for all or some of our messages that stuff their data directly into a buffer, saving us CPU time. And if we get smarter in a different way, we can create automatic bit-packing serialization. Because we control the grammar for our message definitions, we can even enable options like these on a message-by-message or protocol-by-protocol basis. The important thing with optimization is to do it smartly - only when profiling of CPU or bandwidth shows that it's necessary. But the nice thing about all this generated code is that if it's necessary for one protocol or message, every protocol or message can benefit.

Fallback to JSON Serialization

- Switch to JSON serialization when binary CRC check fails
- Great for programmers
- Way more expensive (CPU and Bandwidth)
- Never allowed on public facing protocols
- Even internally it's sometimes unreasonable



When two services have determined that they have binary incompatible versions of a protocol,
[ADVANCE] This triggers a switch to JSON serialized JAM.
[ADVANCE] JSON is great for programmers. It's easy to look at anything encoded in JSON and see exactly what's in it.
[ADVANCE] Unfortunately it's way more expensive in bandwidth and CPU than our binary protocol.
[ADVANCE] So we'll never allow JSON fallbacks on a public facing protocol,
[ADVANCE] and even for some internal tools it isn't entirely reasonable.
[DO NOT ADVANCE]

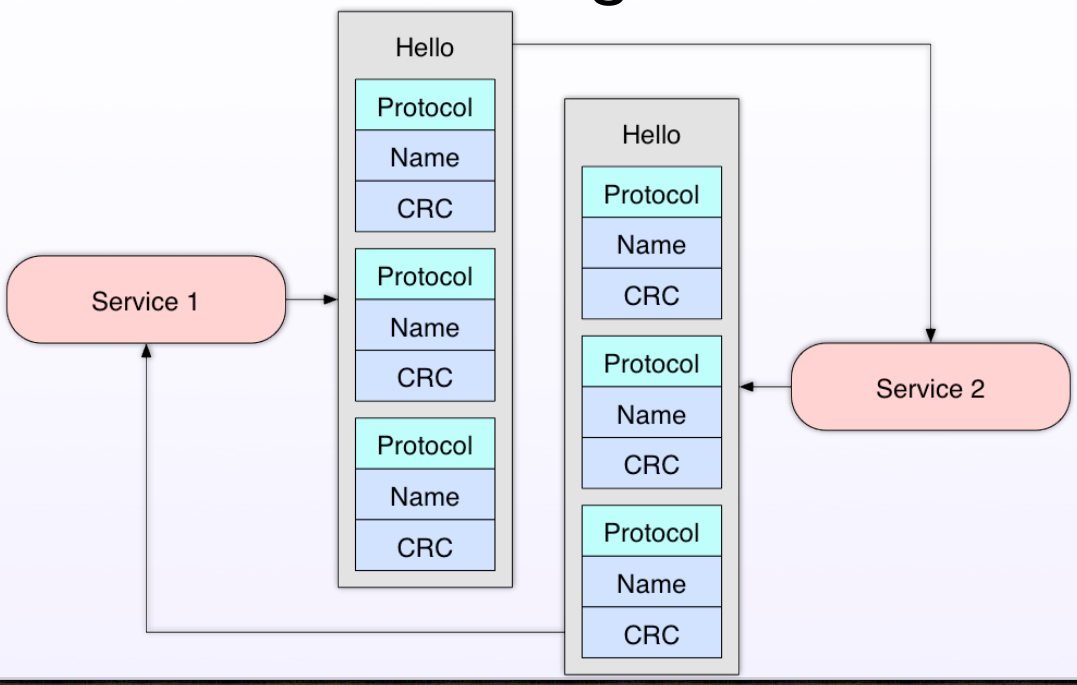
But because all the data is sent using name=value pairs in plain text, it makes it very easy to take data sent from a different version of a protocol and fill out a structure in the current version. If there are fields that aren't present, they can be set to default values, and if there are fields we don't know about, they can be safely ignored. Even if fields have changed types, we can still deal with it in many cases.

JSON

```
{
    "_msgID":10,
    "type":6,
    "error":0,
    "desc":{
        "m_id":"T2R00S40.00E14815726P10987H127.0.0.1:14001",
        "m_host":"127.0.0.1",
        "m_partitionID":0,
        "m_configID":0,
        "m_buildNum":0,
        "m_type":40,
        "m_subType":0
    }
}
```

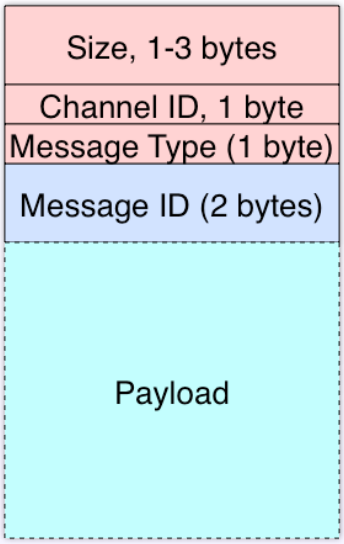
In case anyone doesn't know what I'm talking about, this is what JSON looks like. It stands for Javascript Object Notation, and it's just a way of expressing data as dictionaries and arrays. It's fantastic for humans, but not so great for computers compared to the binary representation when parsing speed and bandwidth matter.

Protocol Negotiation

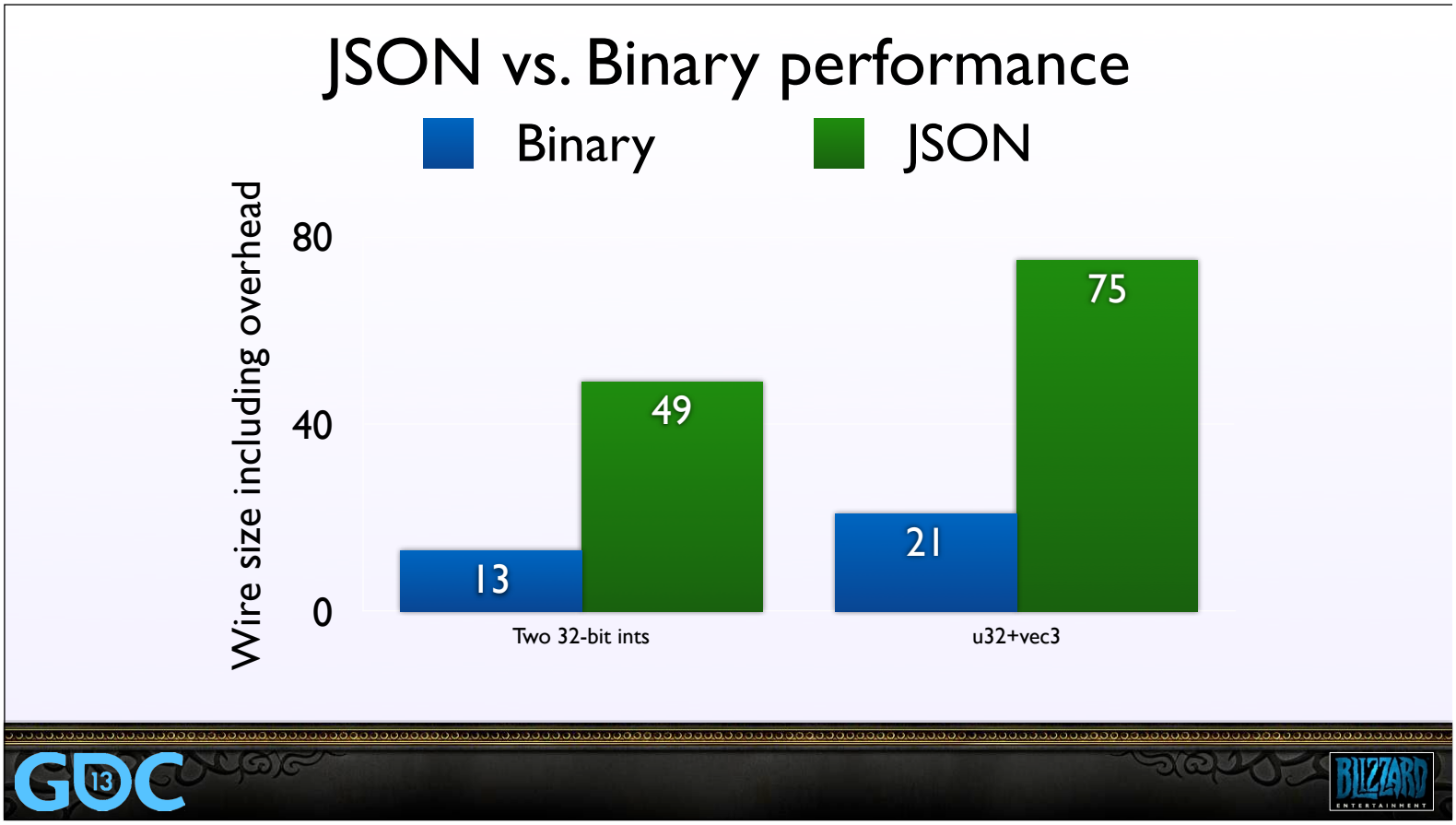


For our game client in particular, we enforce a matching protocols only rule. If the protocol negotation that happens at link-up determines that the other end doesn't match on any protocols, we treat it as an error and don't allow the connection. This functionality is exposed as a flag passed into RouteConfig. On the slide, you can see how JAM internally figures out protocols and channels. When a new connection is established, each end immediately sends a list of all the protocols it supports with names, CRCs, and whether it's willing to send or receive, or both, for each protocol. Its peer then checks to see if it's willing to receive each message the other end is willing to send, and vice versa. Each protocol where a send/receive match is found is then deterministically assigned a one byte channel ID (yes, we are limited to 256 active protocols on a given connection. I hope that never becomes an issue!). The ID is based on the order in which the protocols were found in the metadata, and we use the listen vs. connect status of a connection to determine which metadata takes precedence. In future communications, that channel ID is used to detemine which protocol each message sent belongs to.

Message overhead

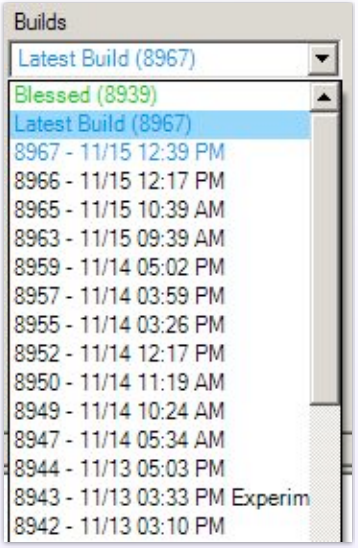


Our overhead for sending a message is a size (1-3 bytes), one byte for a channel ID, one byte for a message type, and two bytes for a message ID within the protocol defined by that channel. In the case of an object message there's also a 128-bit object ID. I will note that we don't use object messages on public facing protocols, only on internal ones. The message type is always either a service message or an object message for application messages. Internally JAM has a few more types for protocol negotiation, pings, and other connection management.



Having the ability to run services with different versions of the same protocol has allowed us to create an environment where we are not locked into synchronized updates. We have an asset pipeline with many custom tools and services, and we are free to add features at the protocol level at any time. The downside of this is that when we deploy a new version of a service that implements a high volume protocol, performance may suffer significantly. There are only a handful of cases where it really matters, but it's almost too easy to make a change in one of those cases without realizing it. You can see from this chart that even for small, simple messages, JSON serialized messages can be more than 3 times larger than binary.

Still highly successful - some network tools run on old versions frequently

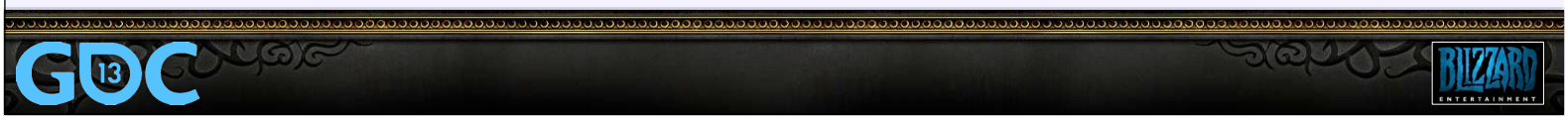


Despite the drawbacks to JSON fallback, it has been a highly successful means of keeping our build process sane. When we have new features in our back-end servers that require protocol changes, we are able to make those changes, deploy new versions, and keep everyone working without downtimes or required updates. Though we have a good tools deployment process, we're also a project still under heavy development, and letting people stick with old versions is sometimes important. We're averaging 11 builds a day, and sometimes "latest" is not "greatest".

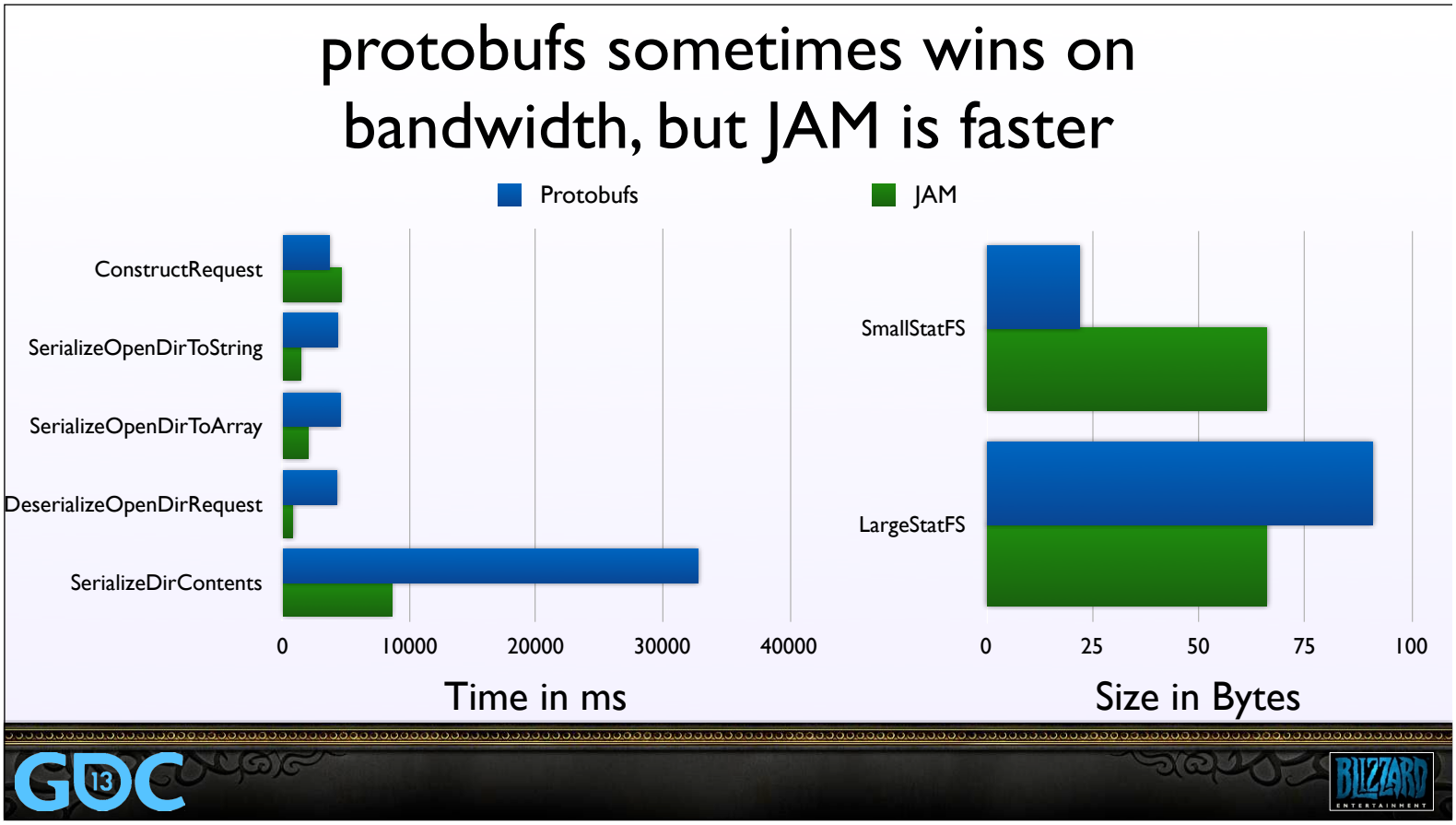
Google's Protocol Buffers and

I've talked a lot about how we solved this problem. We're not the only ones to take it on though. Back in 2001 when I started working on JAM, there wasn't a good alternative that I knew about. Luckily for you, though, there are now several. Google Protocol Buffers is one popular choice, and it's even been used for some projects at Blizzard. Facebook's Thrift is another good one, as is Apache's Avro. If you're struggling with serialization issues, you could do much worse than any of those projects.

For both speed AND inter-version compatibility, there are better choices



All three of those projects offer a binary serialization protocol that also solves the problem of inter-version compatibility. What this means is that they're all relatively bandwidth efficient while still allowing you to deploy newer versions of your protocol without fear of breaking things. Their priorities are a little bit different than JAM's historical purpose. JAM was intended first and foremost to be a highly efficient protocol between servers where version-matching was a given. Protobufs and the others were designed to be used in environments where it was a given that there would be services that never matched and might even be years out of date in some cases. I think for many of us here today, working on games where speed still matters, a JAM-like approach is still appropriate, and I don't actually know of a good open-source alternative with the same priorities as JAM.



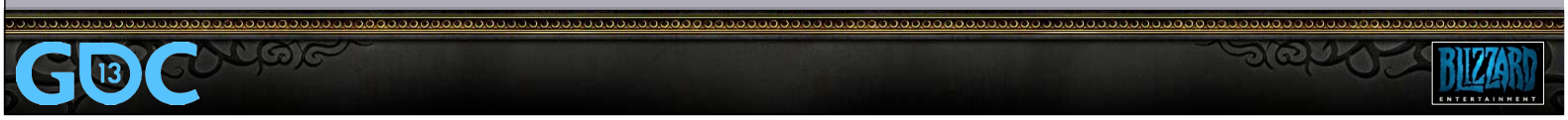
These are some benchmarks of equivalent protobufs and JAM messages.

The left chart here are times in milliseconds for some typical serialization and deserialization tasks. The right hand chart shows byte sizes for one message in two different scenarios.

Even though protobufs has a small overhead beyond the pure binary data in order to provide enough metadata about serialized messages for different versions of the same protocol to work, it can actually be more bandwidth efficient than JAM. This comes down to smart integers – protobufs implements a method of sending integers where smaller integers take fewer bits to send. For example, a 32 bit integer whose actual value is 5 will only take up one byte on the wire. JAM doesn't do any packing of that sort, but it's actually something I'm looking at adding (TODO: might have it by March, revise this!) In practice, for well built protocols, protobufs doesn't very often achieve that result. A potentially large disadvantage of protobufs though is CPU. In benchmarks of relatively simple messages, we found protobufs default C++ implementation to be 3-5x slower than JAM at serializing and deserializing data in virtually all cases using the most efficient methods protobufs provides. That was somewhat surprising, but I'll note here that there are several alternate implementations that may be faster than Google's. If you are looking into using or setting up a system like JAM, those implementations are definitely worth looking into.

Writing our own gives us ultimate control over everything

- Automated serialization
- Easy yet flexible message dispatching
- High performance
- Inter-version compatibility
- Less tedium = more awesome



When I started writing JAM, I didn't have any alternatives. Although there are some reasonably good partial solutions out there now, there's something to be said for doing it yourself. Though I've only scratched the surface in this talk, a system like this isn't actually all that hard to create. If you are struggling with
[ADVANCE]data serialization
[ADVANCE]or message delivery systems, you owe it to yourself to find a solution.
[ADVANCE] Your generated code can be just as fast, or faster, than hand-rolled code,
[ADVANCE] and you can easily achieve cross-version compatibility.
[ADVANCE] All of this means we spend less time on the boring parts of network programming and on debugging and more time on making our game awesome.

Thanks! Questions?

Joe Rumsey
jrumsey@blizzard.com
Twitter: @joerumz

Disclaimer: Blizzard is not really making World of Checkers

Thanks for coming
Hope you got something out of this
Happy to answer questions by email or twitter
[ADVANCE FOR DISCLAIMER]