Building blocks for a

Minimal RPC framework

with modern C++

Rui Figueira

https://github.com/ruifig/czrpc
http://www.crazygaze.com

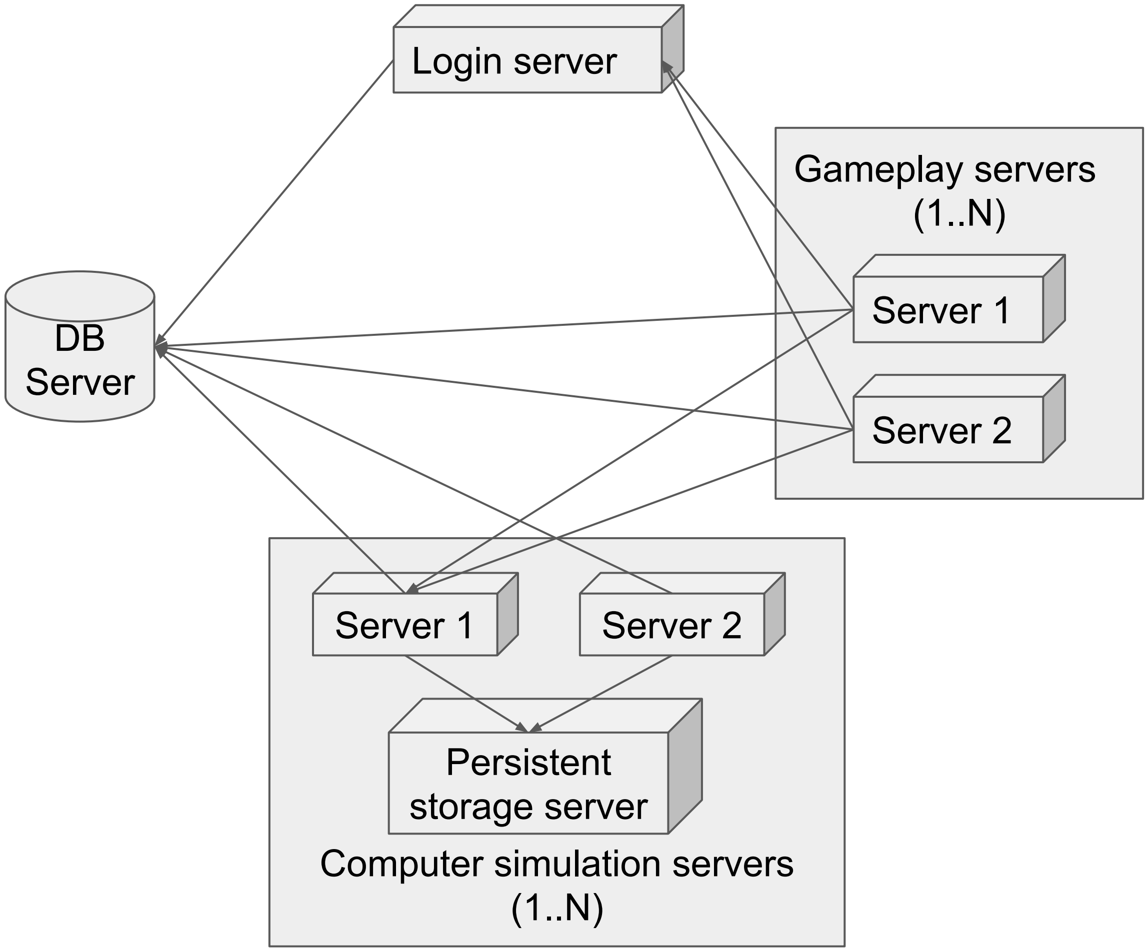
Why ?

- Experiment with C++11/14 features
  - Variadic templates
  - Lambdas
  - Move semantics
  - Auto type deduction
  - decltype
- Can I do away with service definition files?
- … and still have an acceptable API ?
- Tailored for my needs

My needs

- Multiple servers
- Backend only
- C++
- Binary serialization
- Type rich
- Trusted peers

https://bitbucket.org/ruifig/g4devkit



Features

- Type-safe
- No service definition files
- Simple API
- Not limited to pre-determined types
- Bidirectional RPCs
- Two ways to handle RPC results
- Non intrusive
- Header-only
- No external dependencies
- … and more ...

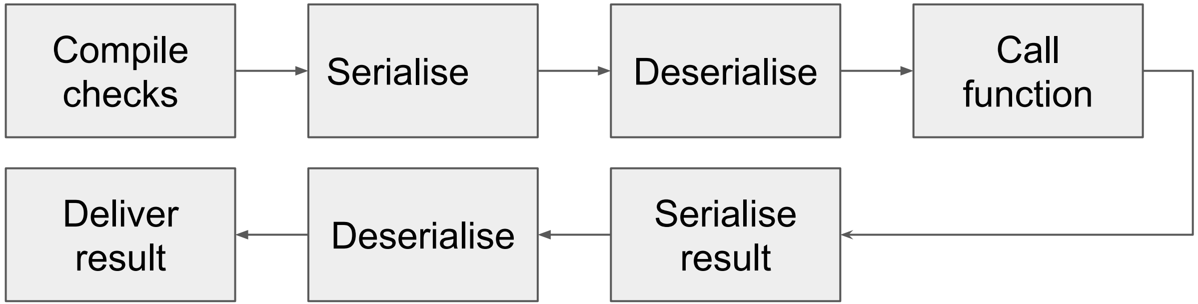
Complete example

- Interface definition
- Server
- Client
- 1 RPC call handled with std::future

```cpp
// Server interface
class Calc {
 public:
  int add(int a, int b) { return a + b; }
  int sub(int a, int b) { return a - b; }
};
#define RPCTABLE_CLASS Calc
#define RPCTABLE_CONTENTS \
  REGISTERRPC(add)        \
  REGISTERRPC(sub)
#include "crazygaze/rpc/RPCGenerate.h"

void testSimpleServerAndClient() {
  // Calc server on port 9000
  Calc calc;
  RPCServer<Calc> server(calc, 9000);
  // Client and 1 RPC call (Prints '3')
  RPCClient<void, Calc> client("127.0.0.1", 9000);
  std::cout << CZRPC_CALL(client->con, add, 1, 2).ft().get().get();
}
```

Problems

- Return and parameter types (type safety)
  - Can a function be used for RPCs (parameters and return type are of acceptable types) ?
  - Supplied arguments match (or are convertible) to what the function expects ?
- Serialize
- Deserialize
- Call the desired function



Checking a function signature with a known number of parameters

```
// Check if "T" is an arithmetic type
static_assert(std::is_arithmetic<T>::value, "");  // C++11-14
static_assert(std::is_arithmetic_v<T>, "");       // C++17

// Check if a function signature "R(A)" only uses arithmetic types
template <class F>
struct FuncTraits {};
template <class R, class A>
struct FuncTraits<R(A)> {
  static constexpr bool valid = std::is_arithmetic_v<R> && std::is_arithmetic_v<A>;
};

int func1(int a);  // Valid signature
static_assert(FuncTraits<decltype(func1)>::valid, "Has non-int parameters");
void func2(std::string a);  // Invalid signature
static_assert(FuncTraits<decltype(func2)>::valid, "Has non-int parameters");
```

Supporting arbitrary types: What do we need to know?

What we need to know from a parameter type?

- Is it a valid type?
- How do we represent it as an Ivalue ?
  - Because when deserializing, we need a variable to deserialise it to.
- How do we serialize it ?
- How do we deserialise it ?
- Given the deserialised value (into an Ivalue), how do we make it into a valid argument for the function?

ParamTraits<T>

```
template <T>
struct ParamTraits {
  // Valid as an RPC parameter ?
  static constexpr bool valid = ???;

  // Type to use for the lvalue when deserialising
  using store_type = ???;

  // Serialize to a stream (v not necessarily of type T)
  template <typename S>
  static void write(S& s, T v);

  // Deserialise from a stream
  template <typename S>
  static void read(S& s, store_type& v);

  // Returns what to pass to the rpc function
  static T get(store_type&& v);
};
```

Supporting arbitrary types: Arithmetic types

```cpp
// By default all types are invalid
template <typename T, typename ENABLE = void> struct ParamTraits {
  static constexpr bool valid = false;
  using store_type = int;
};

// Support for any arithmetic type
template <typename T>
struct ParamTraits<T, typename std::enable_if_t<std::is_arithmetic_v<T>>> {
  static constexpr bool valid = true;
  using store_type = T;
  template <typename S> static void write(S& s, T v) {
    s.write(&v, sizeof(v));
  }
  template <typename S> static void read(S& s, store_type& v) { s.read(&v, sizeof(v)); }
  static store_type get(store_type v) { return v; }
};

static_assert(ParamTraits<int>::valid == true, "Invalid"); // OK
static_assert(ParamTraits<double>::valid == true, "Invalid"); // OK
// No refs allowed by default (can be tweaked later)
static_assert(ParamTraits<const int&>::valid == true, "Invalid"); // ERROR
```

Supporting arbitrary types: Extending to support const T&

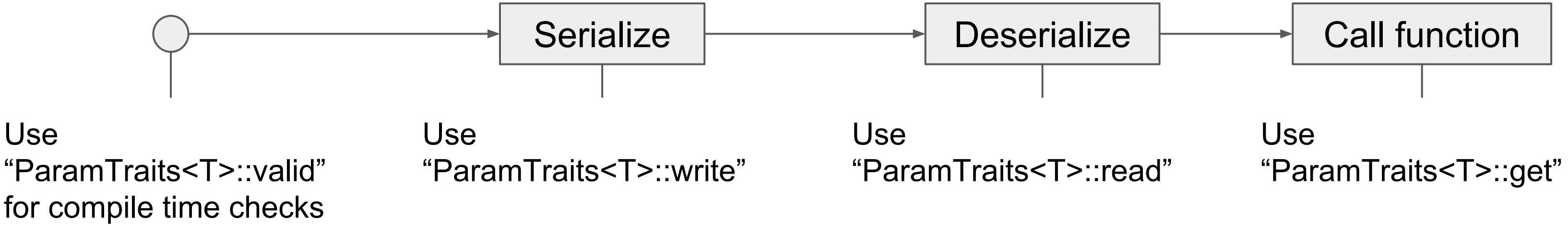
```
// Explicit specialization for const int&
template <> struct ParamTraits<const int&> : ParamTraits<int> {};

// Generic support for const T&, for any valid T
template <typename T> struct ParamTraits<const T&> : ParamTraits<T> {
  static_assert(ParamTraits<T>::valid, "Invalid RPC parameter type");
};

static_assert(ParamTraits<const int&>::valid == true, ""); // OK
static_assert(ParamTraits<const std::string&>::valid == true, ""); // Error
static_assert(ParamTraits<const double&>::valid == true, ""); // OK
```

Supporting arbitrary types: Non-arithmetic types

```
template <typename T> struct ParamTraits<std::vector<T>> {
  using store_type = std::vector<T>;
  static constexpr bool valid = ParamTraits<T>::valid;
  static_assert(ParamTraits<T>::valid == true, "T is not valid RPC parameter type.");

  // Write the vector size, followed by  each element
  template <typename S> static void write(S& s, const std::vector<T>& v) {
    unsigned len = static_cast<unsigned>(v.size());
    s.write(&len, sizeof(len));
    for (auto&& i : v) ParamTraits<T>::write(s, i);
  }

  template <typename S> static void read(S& s, std::vector<T>& v) {
    unsigned len;
    s.read(&len, sizeof(len));
    v.clear();
    while (len--) {
      T i;
      ParamTraits<T>::read(s, i);
      v.push_back(std::move(i));
    }
  }

  static std::vector<T>&& get(std::vector<T>&& v) { return std::move(v); }
};
```

Supporting arbitrary types: Why we need store_type

```cpp
// Hypothetical serialisation functions
template <typename S, typename T> void serialize(S& s, const T& v) { /* ... */ }
template <typename S, typename T> void deserialise(S&, T&) { /* ... */ }

void test_serialization() {
  Stream s;
  // T=int shows no problems
  int a = 1;
  serialize(s, a);
  deserialise(s, a);
  // How about T=const char*
  const char* b = "Hello";
  serialize(s, b);
  deserialise(s, b); // You can't deserialise to a const char*
}
```



Supporting arbitrary types: const char*

```
// Barebones for const char* support
template <> struct ParamTraits<const char*> {
  static constexpr bool valid = true;
  using store_type = std::string;
  template <typename S> static void write(S& s, const char* v) { /* ... */ }
  template <typename S> static void read(S& s, store_type& v) { /* ... */ }
  // Convert to what the function really expects
  static const char* get(const store_type& v) { return v.c_str(); }
};
```

- We use an std::string for deserialisation, then “get” converts to the right parameter type.
- Similar specializations can be made for char[N], const char[N], etc

Function traits: Making use of ParamTraits

- We now know what we need about valid types
- Now, given a function signature, we need a similar FuncTraits<F> that collects all relevant information in one place
  - Is the return type and all parameter types valid ?
  - How do we serialize all arguments ?
  - How do we unserialize them in way we can use them to call the function

```
template <typename F> struct FuncTraits {
  using return_type = ??? ;
  using param_tuple = std::tuple <???> ;
  static constexpr bool valid = ??? ;
  static constexpr std::size_t arity = ??? ;
  // Get a parameter type by its index
  // ...
};
```

Function traits: Checking all parameters for validity

Helper variadic template class to check ParamTraits on N parameters…

```
template <typename... T>
struct ParamPack {
  static constexpr bool valid = true;
};
template <typename First>
struct ParamPack<First> {
  static constexpr bool valid = ParamTraits<First>::valid;
};
template <typename First, typename... Rest>
struct ParamPack<First, Rest...> {
  static constexpr bool valid = ParamTraits<First>::valid && ParamPack<Rest...>::valid;
};
```

FuncTraits for methods

```
template <class F> struct FuncTraits {};

// method pointer
template <class R, class C, class... Args>
struct FuncTraits<R (C::*)(Args...)> : public FuncTraits<R(Args...)> {
  using class_type = C;
};

// const method pointer
template <class R, class C, class... Args>
struct FuncTraits<R (C::*)(Args...) const> : public FuncTraits<R(Args...)> {
  using class_type = C;
};
```

Function traits: FuncTraits details

```
template <class R, class... Args>
struct FuncTraits<R(Args...)> {
  using return_type = R;
  static constexpr bool valid =
      ParamTraits<return_type>::valid && ParamPack<Args...>::valid;
  using param_tuple = std::tuple<typename ParamTraits<Args>::store_type...>;
  static constexpr std::size_t arity = sizeof...(Args);

  template <std::size_t N>
  struct argument {
    static_assert(N < arity, "error: invalid parameter index.");
    using type = typename std::tuple_element<N, std::tuple<Args...>>::type;
  };
};
```

```
// How to use ...
using Traits = FuncTraits<decltype(&Foo::f1)>;
static_assert(Traits::valid, "");
Traits::argument<0>::type p0; // Get the type of the first parameter
```

Serialize all parameters of a function call

```cpp
template <typename F, int N>
struct Parameters {

  template <typename S>
  static void serialize(S&) {}

  template <typename S, typename First, typename... Rest>
  static void serialize(S& s, First&& first, Rest&&... rest) {
    using Traits = ParamTraits<typename FuncTraits<F>::template argument<N>::type>;
    Traits::write(s, std::forward<First>(first));
    Parameters<F, N + 1>::serialize(s, std::forward<Rest>(rest)...);
  }
};
```

Serialize all parameters of a function call : Example

```
struct Foo {
  float bar(float a, int b) { return a * b; }
};

void testSerializeParameters()
{
    rpc::Stream s;
    // Ok : All arguments match what Foo::bar expects
    Parameters<decltype(&Foo::bar), 0>::serialize(s, 1.0f, 2);
    // Error : First argument for Foo::bar is wrong. It should be a float
    Parameters<decltype(&Foo::bar), 0>::serialize(s, "Hello", 2);
}
```

Deserialize all parameters into a tuple

```cpp
template <typename... T>
struct ParamTraits<std::tuple<T...>> {
  using tuple_type = std::tuple<T...>;
  using store_type = tuple_type;
  static constexpr bool valid = ParamPack<T...>::valid;

  static_assert(ParamPack<T...>::valid == true,
      "One or more tuple elements are not of valid RPC parameter types.");

  template <typename S>
  static void write(S& s, const tuple_type& v) {
    details::Tuple<tuple_type, std::tuple_size<tuple_type>::value == 0, 0>::serialize(s, v);
  }

  template <typename S>
  static void read(S& s, tuple_type& v) {
    details::Tuple<tuple_type, std::tuple_size<tuple_type>::value == 0, 0>::deserialize(s, v);
  }

  static tuple_type&& get(tuple_type&& v) { return std::move(v); }
};
```

Deserialize all parameters into a tuple: Example

```
struct Foo {
  float bar(float a, int b) { return a * b; }
};

void testSerializeParameters() {
  rpc::Stream s;

  // Serialise
  Parameters<decltype(&Foo::bar), 0>::serialize(s, 1.0f, 2);

  // Deserialise
  FuncTraits<decltype(&Foo::bar)>::param_tuple params;
  ParamTraits<decltype(params)>::read(s, params);
}
```

Call a function given a tuple of parameters

- Similar to std::apply
  - std::apply does **somefunc(std::get<0>(t), std::get<1>(t), ...)**
- We need to transform from a ParamTraits<N>::store_type into the expected type
  - **somefunc(ParamTraits< FuncTraits<F>::argument<N>::type > ::get(**std::get<N>(t)**))**

For example, consider a parameter of const char* type. “store_type” for this type would be std::string :

```
void somefunc(const char* p0);
std::tuple<std::string> params;  ← Has the deserialized parameters

What we need to pass to the function call...
somefunc( ParamTraits<const char*>::get( std::get<0>(params) ) )
```

Call a function given a tuple of parameters: Implementation

```
namespace detail {
template <typename F, typename Tuple, size_t... N>
decltype(auto) callMethod_impl(
    typename FuncTraits<F>::class_type& obj, F f, Tuple&& t, std::index_sequence<N...>) {
  return (obj.*f)(ParamTraits<typename FuncTraits<F>::template argument<N>::type>::get(
      std::get<N>(std::forward<Tuple>(t)))...);
}
}  // namespace detail

template <typename F, typename Tuple>
decltype(auto) callMethod(typename FunctionTraits<F>::class_type& obj, F f, Tuple&& t) {
  static_assert(FunctionTraits<F>::valid, "Function not usable as RPC");
  return detail::callMethod_impl(
      obj, f, std::forward<Tuple>(t), std::make_index_sequence<FuncTraits<F>::arity>{});
}
```

Call a function given a tuple of parameters: Example

```
void testCallMethod() {
  rpc::Stream s;

  // Serialise
  Parameters<decltype(&Foo::bar), 0>::serialize(s, 1.0f, 2);

  // Deserialise
  FuncTraits<decltype(&Foo::bar)>::param_tuple params;
  ParamTraits<decltype(params)>::read(s, params);

  // Call method given a tuple
  Foo foo;
  auto res = callMethod(foo, &Foo::bar, std::move(params));
}
```

RPC Table:

So, back to what we saw at the beginning, what is this?

```cpp
// Server interface
class Calc {
 public:
  int add(int a, int b) { return a + b; }
  int sub(int a, int b) { return a - b; }
};
#define RPCTABLE_CLASS Calc
#define RPCTABLE_CONTENTS \
  REGISTERRPC(add)        \
  REGISTERRPC(sub)
#include "crazygaze/rpc/RPCGenerate.h"
```

RPC Table : Barebones

```
template <typename T> class Table { static_assert(sizeof(T) == 0, "RPC Table not specified for the type."); };
// The macros generate a specialisation Table<Calc>
template <> struct Table<Calc> {
  using Dispatcher = std::function<void(Calc&, Stream&, Stream&)>; // Using type erasure to handle differences
  enum class RPCId { add, sub };
  std::vector<Dispatcher> m_dispatchers;  // The index is the RPC id

  Table() {
    registerRPC(&Calc::add);
    registerRPC(&Calc::sub);
  }

  template <typename F>
  void registerRPC(F f) {
    auto dispatcher = [f](Calc& obj, Stream& in, Stream& out) {
      typename FuncTraits<F>::param_tuple params;
      in >> params;
      out << callMethod(obj, f, std::move(params));
    };
    m_dispatchers.push_back(std::move(dispatcher));
  }
};
```

CZRPC_CALL macro

```cpp
std::cout << CZRPC_CALL(client->con, add, 1, 2).ft().get().get();
```

We can figure out it is Calc type

This is both the RPC Id, and the method name. We know the function, so we know how to serialize all the parameters

Parameters

Result<int>::get() to get the call result

Actual RPC result (Result<int>) Result<T> wraps the result and deals with success/exceptions/disconnects

CZRPC_CALL puts together the data ready to send (header + payload), but doesn't send.
The user then uses .ft() or .async(...) to trigger the send and Specify how to handle the result.

```cpp
// In steps...
auto call = CZRPC_CALL(client->con, add, 1, 2); // Does type checks and packs all the data
std::future<Result<int>> resFt = call.ft(); // Submit the call and handle it with an std::future
Result<int> res = resFt.get(); // Wait until we receive the result
int val = res.get(); // Get the result. Result<T> has other functions to check for errors, exceptions, etc
```

CZRPC_CALL : Asynchronous handlers

```
// Asynchronous handling
CZRPC_CALL(client->con, add, 1, 2).async([](Result<int> res)
{
    std::cout << res.get();
});
```

Things not explained

- Class to serialize a call, send it, and handle replies
- Class that receives data, calls the appropriate Table<T> dispatcher, and sends back the result
- Most of the missing blocks are built with what was already presented here.
- Lifetime management
    - User needs to make sure relevant objects stay valid while there are pending replies using them.
    - Framework doesn’t guess. It’s up to the user and/or transport

Future improvements

- Keep using it as a testbed for creating RPC frameworks without service definition files
    - Upcoming C++ reflection
    - Herb Sutter’s proposed metaclasses
        - https://www.youtube.com/watch?v=6nsyX37nsRs
- Default transport uses https://github.com/ruifig/czspas
    - Pro: Small simple Asio like API
    - Con: Potentially adds some latency
- Create other transports

More info

- https://github.com/ruifig/czrpc
  - This RPC framework
- https://github.com/ruifig/czspas
  - Small Portable Asynchronous Sockets (Asio like API)
  - Used as the default transport in czrpc
- http://www.crazygaze.com/
  - Personal website, with C++ articles
- https://bitbucket.org/ruifig/g4devkit
  - Devkit for the game in development (will move to GitHub soon)
- https://www.jetbrains.com/clion/
  - I’m using a FREE Open Source license to work on Linux.
  - Coincidentally, our next talker (Phil Nash) works at JetBrains :)