Subspace IPC

Next Generation, extremely high-performance shared memory IPC

David Allison

This is a publish/subscribe message passing system with sub-microsecond message latency on the same computer, regardless of message size. It allows for extremely fast communication between processes and is aimed at robotics applications where speed is vital, and memory is limited.

The simplest type of IPC system uses TCP or UDP to send messages between processes, and a lot of projects start and stop right there. Modern operating systems have a very nicely tuned network stack so it's a reasonable choice.

However, if you are sending messages between processes on the same computer, using the whole network stack to copy from process memory, through the kernel and back into another process's memory is a bit of a waste of CPU time. It would be much better to allocate some shared memory and map it into each process, then just put the message in the shared memory. Each process would see the same data, albeit at a different virtual address.

But, due to the aforementioned tuning, sending small messages using TCP is very fast nowadays, especially if you use the loopback network stack. When the messages get bigger, like high resolution camera images several megabytes each, the memory copies become onerous, and the CPU spends a lot of its time just copying data around.

Why *subspace*? This is a term from sci-fi where you can communicate faster than the speed of light instantaneously across the galaxy. If you consider normal network communication via sockets to be the speed of light, then using subspace (shared memory) you can send any large message in under 1 microsecond (about 250 nanoseconds is the fastest I've measures on a Mac M1). That's faster than the network stack can do it, and thus faster than light. The larger the message the larger the throughput too, so your bandwidth is only limited by memory usage.

This library implements an IPC system that uses shared memory for transport for messages inside a computer, and TCP for messages that go across the computer boundary. Its characteristics include:

1. Single threaded coroutine based server process written in C++17
2. Coroutine-aware client library, in C++17.
3. Publish/subscribe methodology with multiple publisher and multiple subscribers per channel.
4. No communication with server for message transfer.
5. Message type agnostic transmission – bring your own serialization.
6. Channel types meaningful to application.
7. Single lock, triple buffered POSIX shared memory channels
8. Both unreliable and reliable communications between publishers and subscribers.
9. Ability to read the next or newest message in a channel.
10. File-descriptor-based event triggers.
11. Automatic UDP discovery and TCP bridging of channels between servers.

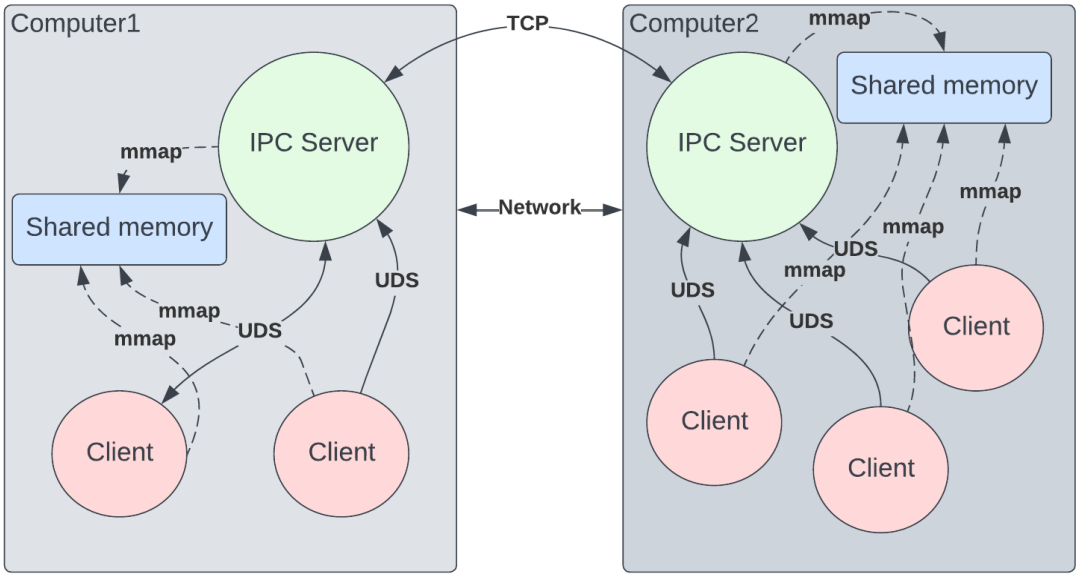
12. Shared and weak pointers for message references.
13. Ports to MacOS and Linux, ARM64 and x86_64.
14. Builds using Bazel and uses Abseil and Protocol Buffers from Google.
15. Uses my C++ coroutine library (https://github.com/dallison/cocpp)

Architecture

This is a server-based system. The server is used to allocate the shared memory and distribute the file descriptors to clients for shared memory and channel events. The intention is that there is a single server running on each computer in the system. The computers need to be able to communicate with each other over a network and that network should support UDP and TCP. Currently only IPv4 is supported.

Clients are implemented as a library that talks to the server over a Unix Domain Socket. This socket is either in the file system or in the abstract namespace if that is supported. There can be many clients per process, but the client library is **not thread-safe**. Instead, it can be used with my coroutine library to share the CPU in a single-threaded process.

The following diagram shows the architecture of the IPC system when there are two computers in the system. Each computer uses shared memory to allow clients to exchange messages and the IPC servers are connected via TCP. Messages destined for subscribers on a different computer are bridged across TCP. The shared memory is allocated by the server on each computer and is mapped into each process using the POSIX *mmap* facility.



The clients are used to create *publishers* and *subscribers* on a named *channel*. A channel consists of a *fixed* number of *fixed-sized slots*, each of which can hold a single message. The publisher specifies the number of slots and the size of each slot when it it created. All publishers on the same channel must agree on the slot size and number of slots. Since this is meant for use

in an embedded system, there is no ability to resize a channel once it's created, since that can result in unbounded memory growth. It's better to size your system based on empirical heuristics than rely on the system growing as necessary.

A channel is unidirectional: publishers publish messages and subscribers see those messages. Publishers and subscribers can be in either reliable or unreliable mode:

1. **Unreliable mode**: a publisher is always able to get a message slot, even if it means that a subscriber will miss a message. This mode is the most common in a robot where you never want to block the publisher and subscribers can deal with missed messages. For example, if a publisher is publishing camera images it really can't stop the camera generating them, but subscribers are usually fine with missing a few camera frames.
2. **Reliable mode:** a subscriber will never miss a message, but a publisher may not be able to get a slot to publish one. This is most useful for Remote Procedure Call (RPC) semantics where you want to guarantee that if a message is sent, the subscribers will actually see that message. It is also useful for back-pressuring publishers earlier in a pipeline to ensure that unnecessary messages are not sent.

Each channel can carry any type of message at any time, but it's best to define a channel to carry a single type of message. The type carried by the channel can be set when the publisher or subscriber is created and can't be changed afterwards. We don't ascribe any meaning to the channel type – it's up to the user of the system to decide what the types are – we just allow an arbitrary string to be used used as the channel type name. For example, if you were to use Google's protobuf as your serialization, you could use the full name of the protobuf type as the type for the channel. This could then be used to look up a database of descriptors that would allow you to deserialize any message on the channel from a command akin to *rosmsg*.

Dependencies

I used the following external dependencies in this library:

1. Google's Bazel to build it (https://github.com/bazelbuild/bazel)
2. Google's Abseil library (https://github.com/abseil/abseil-cpp)
3. Google's Protocol Buffers library (https://github.com/protocolbuffers/protobuf)
4. My C++ Coroutines library (https://github.com/dallison/cocpp)

I've made extensive use of the *absl::Status* and *absl::StatusOr<>* classes in the Client API and in the server. The communication between the client and the server uses Protocol Buffers. The server is single threaded and uses my *coroutines* library to great effect allowing multitasking behavior without the danger and overhead of multithreading.

The Client is coroutine-aware but you do not have to use coroutines.

The Client API

If a process wants to use the IPC system to send or receive messages, it does so by creating an instance of a class called **subspace::Client**, a C++ class provides all the functions necessary to talk to other processes over IPC.

Creating a client

To create a client, make an instance of **subspace::Client**. This takes an optional pointer to a *Coroutine* instance that can be used to share the CPU when making blocking network calls. If the coroutine pointer is *nullptr*, the client will not use coroutines and will block the CPU while taking to the server.

```
  Client(co::Coroutine *c = nullptr) : co_(c) {}
```

Initializing the client

A client is not connected to the server until you call the *Init* function on it. The *Init* function looks like this:

```
// Initialize the client by connecting to the server.
  absl::Status Init(const std::string &server_socket = "/tmp/subspace",
                    const std::string &client_name = "");
```

It takes the name of the Unix Domain Socket that the server is listening to and connects to the server, passing an optional client name. Check the result of the call by calling the function **ok()** on the returned **absl::Status** object. If **ok()** is false, you can get the error returned by calling **ToString().**

For example:

```
  subspace::Client client;
  absl::Status s = client.Init(absl::GetFlag(FLAGS_socket));
  if (!s.ok()) {
    fprintf(stderr, "Can't connect to IPC server: %s\n", s.ToString().c_str());
    exit(1);
  }
```

Creating a publisher

A channel does not exist until there is a publisher, or a subscriber created for it. The publisher establishes the parameters for the channel:

- The number of message slots
- The size of each slot
- Whether to use reliable or unreliable transport for the publisher.
- Whether the channel can be seen across the network.

You can have multiple publishers on a channel, but all must agree on the channel's parameters. If a subscriber is created for a channel before there are any publishers, the channel is created as a

placeholder channel, since the parameters have not yet been established. When the first publisher is created, the channel becomes fully established.

To create a publisher, use the following API:

```
// Create a publisher for the given channel.  If the channel doesn't exit
  // it will be created with num_slots slots, each of which is slot_size
  // bytes long.  The slots will not be resized.
  absl::StatusOr<Publisher *>
  CreatePublisher(const std::string &channel_name, int slot_size, int num_slots,
                  const PublisherOptions &opts = PublisherOptions());
```

The channel name is a string that identifies the channel. All other publishers and subscribers to the channel will use the same name. While any string can be used, it is common practice to name the channel based on the type of message it carries. The slot size is the size of each message slot in bytes. Slots are not resized. The number of slots specifies how many slots there are in the channel. Channels act as ring buffers, so the more slots there are in the channel, the more messages can be stored in the ring buffer until they are recycled. In this regard, an unreliable channel with a large number of slots behaves like a reliable channel as long as the subscribers are able to keep up with the publishers.

The *PublisherOptions* class provides a set of options for the channel. It is defined as the following:

```
class PublisherOptions {
public:
  PublisherOptions() = default;
  PublisherOptions &SetPublic(bool v) {
    public_ = v;
    return *this;
  }
  PublisherOptions &SetReliable(bool v) {
    reliable_ = v;
    return *this;
  }
  PublisherOptions &SetType(std::string type) {
    type_ = std::move(type);
    return *this;
  }

  bool IsPublic() const { return public_; }
  bool IsReliable() const { return reliable_; }
  const std::string &Type() const { return type_; }
```

```
private:
  bool public_ = true;
  bool reliable_ = false;
  std::string type_;
};
```

The options are:

- **Public**: the channel will be made available outside of the current computer and over the network. This is true by default.
- **Reliable**: the publisher may not be able to get a publishing slot if doing so would prevent a subscriber from seeing a message. If there are no subscribers on the channel, a reliable publisher will not be able to send any messages. The default is unreliable, meaning that a publisher will always be able to get a slot for a message, but subscribers may miss some messages if they are too slow.
- **Type**: an identifier for the type carried by the channel. No meaning is ascribed.

The result of *CreatePublisher* is an *absl::StatusOr<Publisher*>* object that either holds an *absl::Status* error or a pointer to a *Publisher* object owned by the client. The *Publisher* object is not part of the public API for the client. Use the *ok()* function on the result of *CreatePublisher* to check if it worked and if it fails, you can see the reason by calling *ToString*() on the *status*(). For example:

```
absl::StatusOr<subspace::Publisher *> pub = client.CreatePublisher(
      "test", 256, num_slots,
      subspace::PublisherOptions().SetPublic(true).SetReliable(reliable));
  if (!pub.ok()) {
    fprintf(stderr, "Can't create publisher: %s\n",
            pub.status().ToString().c_str());
    exit(1);
  }
```

You access the *Publisher* pointer by using the *operator** on the *absl::StatusOr* object. For example:

```
        absl::StatusOr<void *> buffer = client.GetMessageBuffer(*pub);
```

Creating a subscriber

Messages published on a channel can be seen by all subscribers in the channel. An unreliable subscriber may miss some messages and that's probably OK. A reliable subscriber will never miss any messages, but the publisher may not be able to send any other messages until the reliable subscribers catch up. The API to create a subscriber is:

```
// Create a subscriber for the given channel. This can be done before there
  // are any publishers on the channel.
  absl::StatusOr<Subscriber *>
  CreateSubscriber(const std::string &channel_name,
                   const SubscriberOptions &opts = SubscriberOptions());
```

Subscribers to channels can be created before or after publishers and will function the same way regardless of the ordering.

The *SubscriberOptions* class is similar to the *PublisherOptions* class and is defined as:

```
class SubscriberOptions {
public:
  SubscriberOptions() = default;
  SubscriberOptions &SetReliable(bool v) {
    reliable_ = v;
    return *this;
  }
  SubscriberOptions &SetType(std::string type) {
    type_ = std::move(type);
    return *this;
  }

  bool IsReliable() const { return reliable_; }
  const std::string &Type() const { return type_; }


private:
  bool reliable_ = false;
  std::string type_;
};
```

Similar to the *PublisherOptions*, we have *reliable* and the *type*. If reliable is set to true, the subscriber will operate in reliable mode, meaning that it will never miss a message, at the expense of publishers not being able to publish anything as necessary to achieve this goal. A reliable subscriber applies back-pressure onto the publisher and may not be what your application needs.

The type of the message carried by the channel can be set by the subscriber and it must match all other subscribers and publishers on the same channel (if it's set). No meaning is ascribed to the type here.

As an example, here's a subscriber:

```
absl::StatusOr<subspace::Subscriber *> sub = client.CreateSubscriber(
    "test", subspace::SubscriberOptions().SetReliable(reliable));
if (!sub.ok()) {
  fprintf(stderr, "Can't create subscriber: %s\n",
          sub.status().ToString().c_str());
  exit(1);
}
```

Publishing a message from a publisher

To publish a message, you first need a publisher (obviously). Then, you will need somewhere to put your message. The Client has a function *GetMessageBuffer*, that obtains a pointer to the memory into which you can place your message (in an appropriate wire-format). An unreliable publisher always has a buffer available, but a reliable publisher may fail to get one if it would mean that a reliable subscriber would miss a message.

The function is defined as:

```
// Get a pointer to the message buffer for the publisher.  The publisher
// will own this buffer until you call PublishMessage.  The idea is that
// you fill in the buffer with the message you want to send and then
// call PublishMessage, at which point the message will be active and
// subscribers will be able to see it.  A reliable publisher may not
// be able to get a message buffer if there are no free slots, in which
// case nullptr is returned.  The publishers's PollFd can be used to
// detect when another attempt can be made to get a buffer.
absl::StatusOr<void *> GetMessageBuffer(Publisher *publisher);
```

It returns either a pointer to the buffer (as a void*) or an *absl::Status* object explaining why it couldn't get a buffer. If a reliable publisher fails to get a buffer, you will get a *nullptr* returned, not a status, since it's not an error to be unable to get a buffer for a reliable publisher. You will never get a *nullptr* returned for an unreliable publisher.

Once you have your buffer (remember that it's a fixed size buffer, that you specified when you created the publisher), you fill it in, either by serializing a message into it, or some other means. Once it's complete, call the function *PublishMessage* to send it.

```
// Publish the message in the publisher's buffer.  The message_size
// argument specifies the actual size of the message to send.  Returns the
// information about the message sent with buffer set to nullptr since
// the publisher cannot access the message once it's been published.
absl::StatusOr<const Message> PublishMessage(Publisher *publisher,
                                             int64_t message_size);
```

This takes, in addition to the *Publisher* pointer, the actual size of the message being published. Obviously, this must be less than the size of the slot. You get an OK status if all is well. Zero-

sized messages are not allowed. Returns information about the message that was just published, except the buffer pointer (you need to be a subscriber to see that). You can use the timestamp field of the returned *Message* to look up sent messages using the *FindMessage* function (described below).

Reliable publishers

So, you want to send a message to one of more subscribers reliably, meaning that the subscribers will always see the message. This means that it's possible that there isn't an available slot into which you can put your message. In this case, the *GetMessageBuffer* function will return *nullptr*, but what do you do next? You need to know when it's possible to try to send again, like when a slot has become free. You have a couple of choices:

1. Block the program (or thread if you insist on using them) in a loop keeping trying to get a message slot. Best to use a sleep in the loop to avoid spinning the CPU. Not an ideal solution to be sure.
2. Call the function *poll* that will be triggered when the publisher's trigger fd is triggered. You can use the function *GetPollFd(Publisher*)* to get *a struct pollfd* to use for this. The poll call may block the program, or maybe you've structured your program around a central poll loop (which is a good idea).
3. You can call the client function *WaitForReliablePublisher*: a coroutine aware function that will either block the program on a call to poll, or yield control from the coroutine until the publisher can try to send.

Choice #1 isn't great, but depending on how your program is structured, the other choices are fine. They both rely on the fact that a reliable publisher has a trigger file descriptor allocated for it. On Linux this is an *eventfd*, but on other platforms, it's implemented as a *pipe*. The point of this is to allow you to be notified when it might be possible to get a message buffer.

To get the trigger file descriptor you can call *GetPollFd*. This returns a **struct pollfd**, that can be passed directly to a call to *poll*.

```
struct pollfd GetPollFd(Publisher *publisher);
```

As an example, here's how I send reliable messages in my little sample publisher (manual_tests/pub.cc):

```
    for (;;) {
      absl::StatusOr<void *> buffer = client.GetMessageBuffer(*pub);
      if (!buffer.ok()) {
        fprintf(stderr, "Can't get publisher buffer: %s\n",
                buffer.status().ToString().c_str());
        exit(1);
      }
      if (*buffer == nullptr) {
        // Wait for publisher trigger.
```

```
        absl::Status s = client.WaitForReliablePublisher(*pub);
        if (!s.ok()) {
          fprintf(stderr, "Can't wait for publisher: %s", s.ToString().c_str());
          exit(1);
        }
        continue;
      }
      buf = *buffer;
      break;
    }
    size_t len = snprintf(reinterpret_cast<char *>(buf), 256, "%s", "hello");
    absl::StatusOr<const subspace::Message> status =
        client.PublishMessage(*pub, len);
    if (!status.ok()) {
      fprintf(stderr, "Can't publish message: %s\n",
              status.status().ToString().c_str());
      exit(1);
    }
```

In this case, I am blocking the program, since it's just a simple example, but for a more complex one, I'd definitely be using coroutines so that there's no blocking going on.

Receiving a message in a subscriber

Subscribers receive messages published from publishers. To read a message you can call the functions *ReadMessage* or *FindMessage*, defined as the following overloads:

```
// Read a message from a subscriber.  If there are no available messages
// the 'length' field of the returned Message will be zero.  The 'buffer'
// field of the Message is set to the address of the message in shared
// memory which is read-only.  If the read is triggered by the PollFd,
// you must read all the avaiable messages from the subscriber as the
// PollFd is only triggered when a new message is published.
absl::StatusOr<const Message>
ReadMessage(Subscriber *subscriber, ReadMode mode = ReadMode::kReadNext);

// As ReadMessage above but returns a shared_ptr to the typed message.
// NOTE: this is subspace::shared_ptr, not std::shared_ptr.
template <typename T>
absl::StatusOr<shared_ptr<T>>
ReadMessage(Subscriber *subscriber, ReadMode mode = ReadMode::kReadNext);

// Find a message given a timestamp.
absl::StatusOr<const Message> FindMessage(Subscriber *subscriber,
                                          uint64_t timestamp);
```

```
  // AsFindMessage above but returns a shared_ptr to the typed message.
  // NOTE: this is subspace::shared_ptr, not std::shared_ptr.
  template <typename T>
  absl::StatusOr<shared_ptr<T>> FindMessage(Subscriber *subscriber,
                                            uint64_t timestamp);
```

For the normal case where you want to read the next or newest message on a channel, there are two functions available, one to read a message as a *Message* struct, and one to read a message as a *shared_ptr* to a templated type. Let's look at the first one. The *Message* struct is defined as:

```
// This is a message read by ReadMessage.  The 'length' member is the
// length of the message data in bytes and 'buffer' points to the
// start address of the message in shared memory.  The buffer is
// read only.  If there is no message read, the length member will
// be zero and buffer will be nullptr.
struct Message {
  Message() = default;
  Message(size_t len, const void *buf, int64_t ordinal, int64_t timestamp)
      : length(len), buffer(buf), ordinal(ordinal), timestamp(timestamp) {}
  size_t length = 0;
  const void *buffer = nullptr;
  int64_t ordinal = -1;
  int64_t timestamp = 0;
};
```

It contains a pointer to the data for the message (in an appropriate wire format) along with the length of the data, a unique monotonically increasing *ordinal* number and a nanosecond *timestamp* containing the time the message was published. The timestamp is based on the system's monotonic clock and not wall time.

The *ReadMessage* function is always non-blocking and will return a *Message* with *length* 0 and *buffer* as *nullptr* if there is nothing to read.

The *mode* argument for *ReadMessage* specifies what should be read. There are two choices:

1. Read the next message
2. Read the newest message

If you want to see all the messages in the channel, use the *read next* mode (or omit the argument). The *read newest* mode will just give you the latest message to be published on the channel. The enumeration *ReadMode* is defined as:

```
enum class ReadMode {
  kReadNext,
  kReadNewest,
};
```

It is a common operation in a robotics application to only care about the latest state rather than processing all state messages. For example, if a *WorldState* message is being published at 10Hz, say, the receiver doesn't really care about all intervening messages and only needs to know the current state of the world. The *ReadMode::kReadNewest* mode can be used for this type of operation.

The second *ReadMessage* function is very similar in functionality but uses a template to provide a statically typed result in a shared pointer. The shared pointer holds a reference to the message slot in the channel so a publisher will never take the slot from it. This is very useful to hold onto a message when you've read another one from the channel. Be careful though, if you hold too many, you may prevent the publishers from getting a slot to publish a message. The semantics for an *subspace::shared_ptr* are very similar to the STL's *std::shared_ptr*. There is also an *subspace::weak_ptr* class similar to *std::weak_ptr*. See the header file **client.h** for details.

If you want to use the channel as a ring buffer and look up a message that was published at a particular time, you can use the *FindMessage* functions. These take a nanosecond timestamp and perform a fast search of the currently available messages for one sent with that timestamp, returning a Message just like *ReadMessage* does. There's also a *shared_ptr* version for this. If the requested message is found, the subscriber is set to refer to it, relinquishing the reference to any previously held message, which may then be reused. But, how do you know what timestamps to search for? That's where the result of *PublishMessage* comes in. You can use the *timestamp* field of the returned *Message* to determine when it was sent.

That's all good, but how do you know when to call *ReadMessage*? If it's nonblocking you shouldn't just call it in a loop spinning the CPU. Similar to reliable publishers, there is a trigger file descriptor available for all subscribers. This can be obtained by calling the *GetPollFd* function:

```
struct pollfd GetPollFd(Subscriber *subscriber);
```

Similar to the discussion about reliable publishers, your program, depending on its design, can use multiple ways to know when a message can be read. The simplest way is to call *WaitForSubscriber*, which will block the program on a poll, or yield a coroutine:

```
// Wait until there's a message available to be read by the
// subscriber.  If the client is coroutine-aware, the coroutine
// will wait.  If it's not, the function will block on a poll
// until the subscriber is triggered.
absl::Status WaitForSubscriber(Subscriber *subscriber);
```

For a simple program, this is convenient but if I was writing a complex application, I would design it using a central poll loop or use coroutines.

The subscriber's trigger file descriptor is triggered when a new message arrives on the channel. It is cleared when you read the message. This presents a programming problem. If you only read one message when the trigger occurs, you will not get triggered until another one is published, thus your subscriber may miss some messages that are already in the channel.

The solution to this is simple: read all the messages in the channel when the subscriber gets triggered. Something like this, from my simple subscriber example (manual_tests/sub.cc):

```
for (;;) {
    if (absl::Status s = client.WaitForSubscriber(*sub); !s.ok()) {
        fprintf(stderr, "Can't wait for subscriber: %s\n",
                  s.ToString().c_str());
          exit(1);
     }
    for (;;) {
      absl::StatusOr<subspace::Message> msg = client.ReadMessage(*sub);
      if (!msg.ok()) {
        fprintf(stderr, "Can't read message: %s\n",
                msg.status().ToString().c_str());
        exit(1);
      }
      if (msg->length == 0) {
        break;
      }
      int64_t ordinal = client.GetCurrentOrdinal(*sub);
      printf("Message %" PRId64 ": %s\n", ordinal,
             reinterpret_cast<const char *>(msg->buffer));
    }
  }
```

You know that you've reached the end of the messages when you get a *length* of zero.

The server

The server is a program called **subspace_server** in the *server* directory. It's a standalone, single threaded binary that needs to be running before any clients can be used. There is also a C++ class that can be incorporated into other programs as necessary. The Server class is not thread safe and requires my coroutines library.

It is very lightweight, responding to client requests and publishing a couple of channels. It also is responsible for sending channels across the network to other servers, if a channel is public and goes across the computer boundary.

The server listens on a Unix Domain Socket (UDS) for connections from clients. There may be many simultaneous client connections. The server creates the shared memory areas when requested by the clients. The clients map the shared memory into their process.

The server also runs a channel discovery protocol over UDP. This allows the servers to discover the names of the channels published on other computers on the network. The server can be told what network interface to use for the discovery if there is more than one. When a channel needs to be transmitted to another server over the network, the servers create a TCP connection to carry the traffic.

Server command line arguments

The server has defaults for all its arguments, but they can be specified on the command line. The arguments are:

- **--socket**=<socket name>: The name of the UDS to use for client comms. The default is "/tmp/subspace". If you want to run more than one server on the same computer, give each its own socket name.
- **--disc_port**=<port number>: The UDP port number to use for the receiver for the discovery protocol. The server will listen on this UDP port for incoming discovery messages. The default is 6502.
- **--peer_port**=<port number>: The UDP port number to which the discovery messages will be sent. If you have more than one server running on the same machine, the combination of –disc_port and –peer_port can be used to arrange the servers as you wish. The default is the same as the discovery receiver port (6502).
- **--interface**=<interface name>: the name of the network interface (see the output from /sbin/ifconfig) that the server will used for its discovery protocol. If you omit, the server will choose the first interface it sees that supports BROADCAST and has an IPv4 address. Currently only IPv4 is used for discovery and bridging.
- **-log_level**=<level> set the server's log level. Log levels are: "verbose", "debug", "info", "warning", "error", or "fatal". The default is "info" and this flag controls the level at minimum level at which logs will be produced. Set to "debug" to see debug output messages.
- **--local**=<true|false>: False by default, this flag allows the server to not use the discovery or bridge protocols and it will only work on a single computer.

Server channels

The server publishes protocol buffers messages on two channels. The channels are called:

- **/subspace/ChannelDirectory**: all the currently known channel names and parameters
- **/subspace/Statistics**: statistical information for all the currently known channels

The */subspace/ChannelDirectory* messages are send when a new channel is created and when a channel is destroyed when it no longer has any users. Each message is defined in protobuf as:

```
message ChannelInfo {
  string name = 1;
```

```
  int32 slot_size = 2;
  int32 num_slots = 3;
  string type = 4;
}

message ChannelDirectory {
  string server_id = 1;
  repeated ChannelInfo channels = 2;
}
```

That is, the channel directory carries a single message that has a *server_id* (unique name for the server), and a repeated set of *ChannelInfo* messages. This is an unreliable channel so the whole database is sent in every message. The type of the channel directory message is "**subspace.ChannelDirectory**".

The */subspace/Statistics* channel carries the following protobuf messages:

```
message ChannelStats {
  string channel_name = 1;
  int64 total_bytes = 2;
  int64 total_messages = 3;
}

message Statistics {
  string server_id = 1;
  int64 timestamp = 2;
  repeated ChannelStats channels = 3;
}
```

Each message contains the server ID, a nanosecond timestamp and a repeated set of counters for the channels. The receiver can use this information to determine the performance of the IPC system. It's an unreliable channel so all statistics are sent at the same time. It's published every 2 seconds. The type of the statistics channel is "**subspace.Statistics**".

The Server C++ class

The server is implemented as a library containing a C++ class called "*subspace::Server*". The *subspace_server* binary is a main program that uses the library.

The class (in server/server.h) has the following public interface:

```
class Server {
public:
  Server(co::CoroutineScheduler &scheduler, const std::string &socket_name,
         const std::string &interface, int disc_port, int peer_port, bool local);
```

```
  void SetLogLevel(const std::string& level) { logger_.SetLogLevel(level); }
  absl::Status Run();
  void Stop();
};
```

The *Server* class runs inside my *CoroutineScheduler* system (https://github.com/dallison/cocpp) because it's single threaded but performs many tasks at the same time using coroutines, which IMO are much better than threads for this type of system.

The *subsystem_server* program's main.cc contains the implementation for creating and running the server. It's a pretty small program that looks like this:

```
int main(int argc, char **argv) {
  absl::ParseCommandLine(argc, argv);

  co::CoroutineScheduler scheduler;

  g_scheduler = &scheduler;     // For signal handler.
  signal(SIGPIPE, SIG_IGN);
  signal(SIGQUIT, Signal);

  subspace::Server server(
      scheduler, absl::GetFlag(FLAGS_socket), absl::GetFlag(FLAGS_interface),
      absl::GetFlag(FLAGS_disc_port), absl::GetFlag(FLAGS_peer_port),
      absl::GetFlag(FLAGS_local));

  server.SetLogLevel(absl::GetFlag(FLAGS_log_level));
  absl::Status s = server.Run();
  if (!s.ok()) {
    fprintf(stderr, "Error running Subspace server: %s\n", s.ToString().c_str());
    exit(1);
  }
}
```

I trap the SIGQUIT signal and dump all the coroutines in the signal handler. This allows you to get a snapshot of what's going on in the server if it's misbehaving.

Building the system

I use *Bazel* to build this system. The reason I chose it was because I wanted to use Google's Abseil and Protocol Buffers libraries and Bazel makes that so much easier than all other build systems (because Bazel, actually *Blaze,* the internal name, is the build system used to write them in the first place).

Bazel uses BUILD.bazel files to specify what is available to be built and how to build it. It's definitely worth learning to use Bazel although the integration with VSCode is a little lacking in stability at the moment.

To build *Subspace* on Linux, use the following command:

```
$ CC=clang bazel build ...
```

Inside the directory you cloned from github. It will build using G++ but you might see some warnings that *clang* doesn't show by default.

To build on a modern Apple Mac with Apple Silicon:

```
$ bazel --config=apple_silicon build ...
```

You will need to download Apple Xcode and its command line tools for building on a Mac. Apple has docs on how to do that.