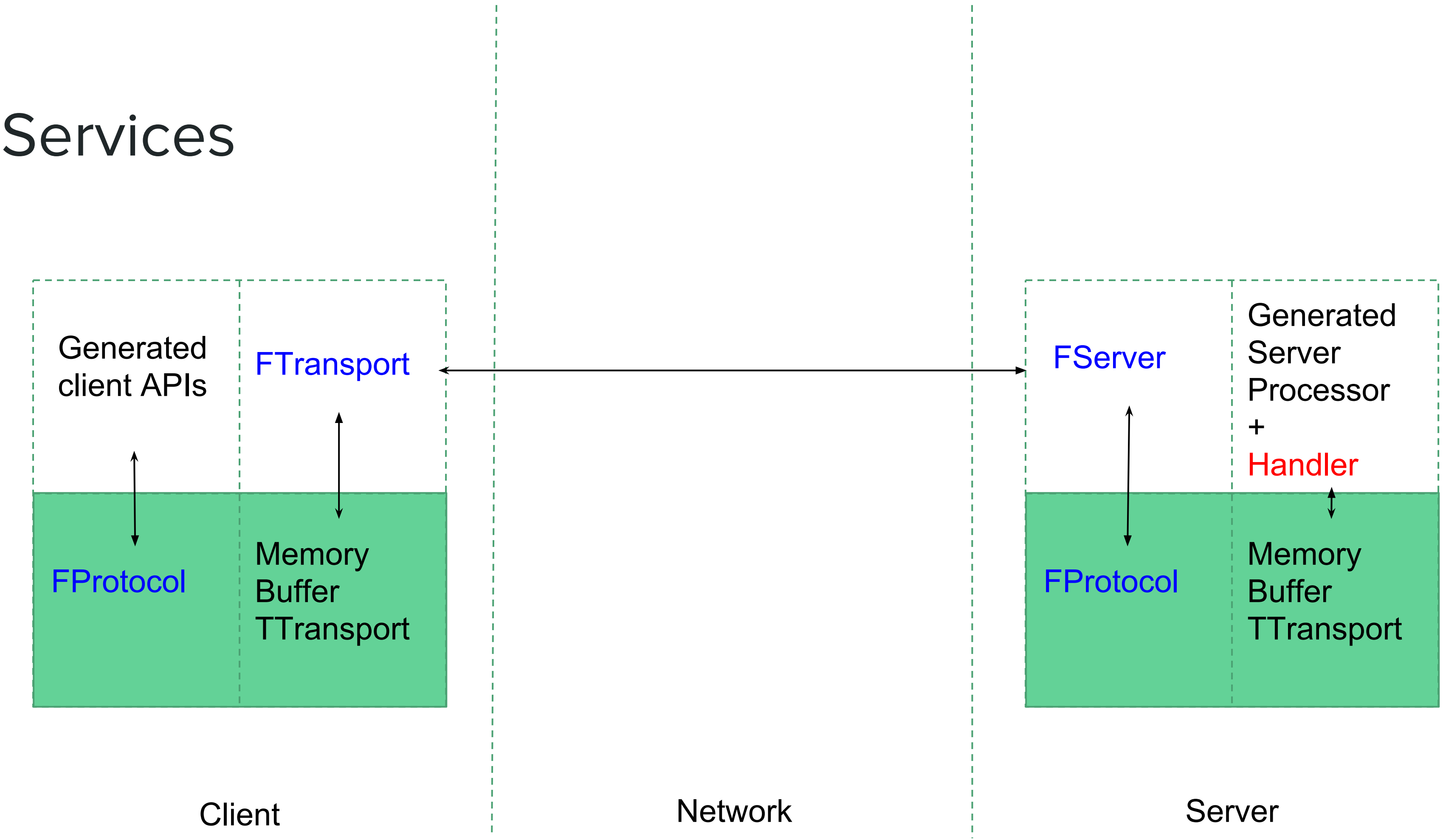
Frugal:
Stack deep dive

Steven Osborne • 12.09.2016

Overview

- High-level
- Protocol
- Transports
- Server
- Generated code

Services



Agreed upon by the client/server

Written by the service implementor

Protocol

From thrift: The Protocol abstraction defines a mechanism to map in-memory data structures to a wire-format. In other words, a protocol specifies how datatypes use the underlying Transport to encode/decode themselves. Thus the protocol implementation governs the encoding scheme and is responsible for (de)serialization.

Jump to java TProtocol interface

FProtocol is an extension of TProtocol that adds the serialization of the FContext object.

Transports

In frugal, we use five distinct entities used to store/transmit data.

1. **TTransport (imported directly from thrift - used for both services and scopes).**
   - In Thrift, the TTransport is the network layer
   - In Frugal, the TTransport is the *buffer* layer
   - Anytime a protocol is encoding/decoding data, it is doing so into/from a TTransport memory buffer
2. **FTransport (service clients)**
   - Network layer for service clients
3. **FServer (service servers)**
   - Server implementations manually wire up receiving network data from clients and route appropriatel server processors (more on this later)
4. **FPublisherTransport (scope publishers)**
   - Network layer for scope publishers
5. **FSubscriberTransport (scope subscribers)**
   - Network layer for scope subscribers

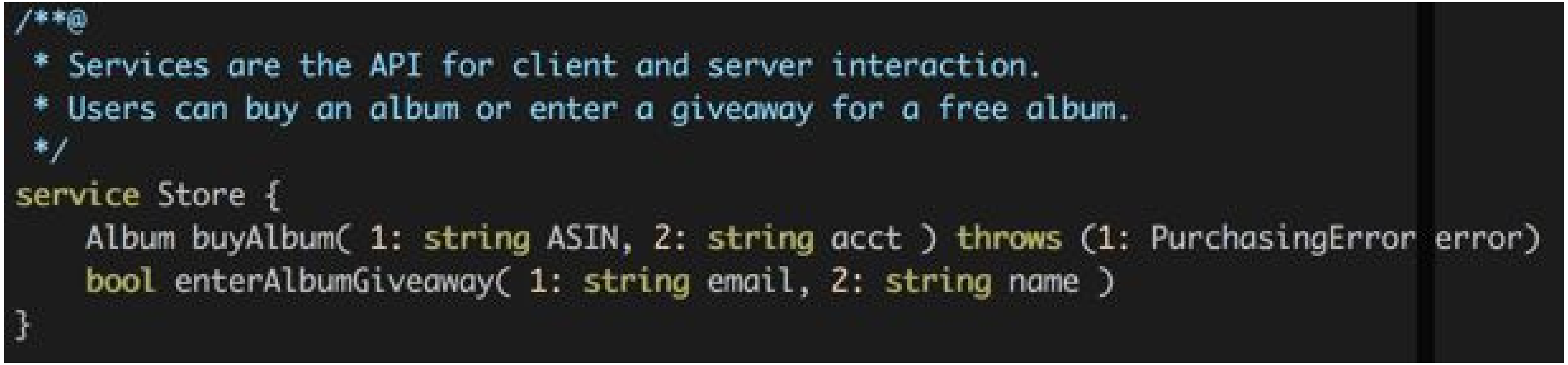
Client

There are two basic components to the client

1. Client API - generated by the frugal compiler
    - serializes the memory objects and hands them to the FTransport to be sent over the wire
    - adds callback routing to be invoked by the FTransport for RPC responses
    - deserializes the the wire-level bytes into memory objects (in the callback)
    - routes objects back to the caller (callback thread -> calling thread)
2. FTransport - interface
    - sends serialized frugal requests to the server
    - Invokes appropriately generated callback upon server response

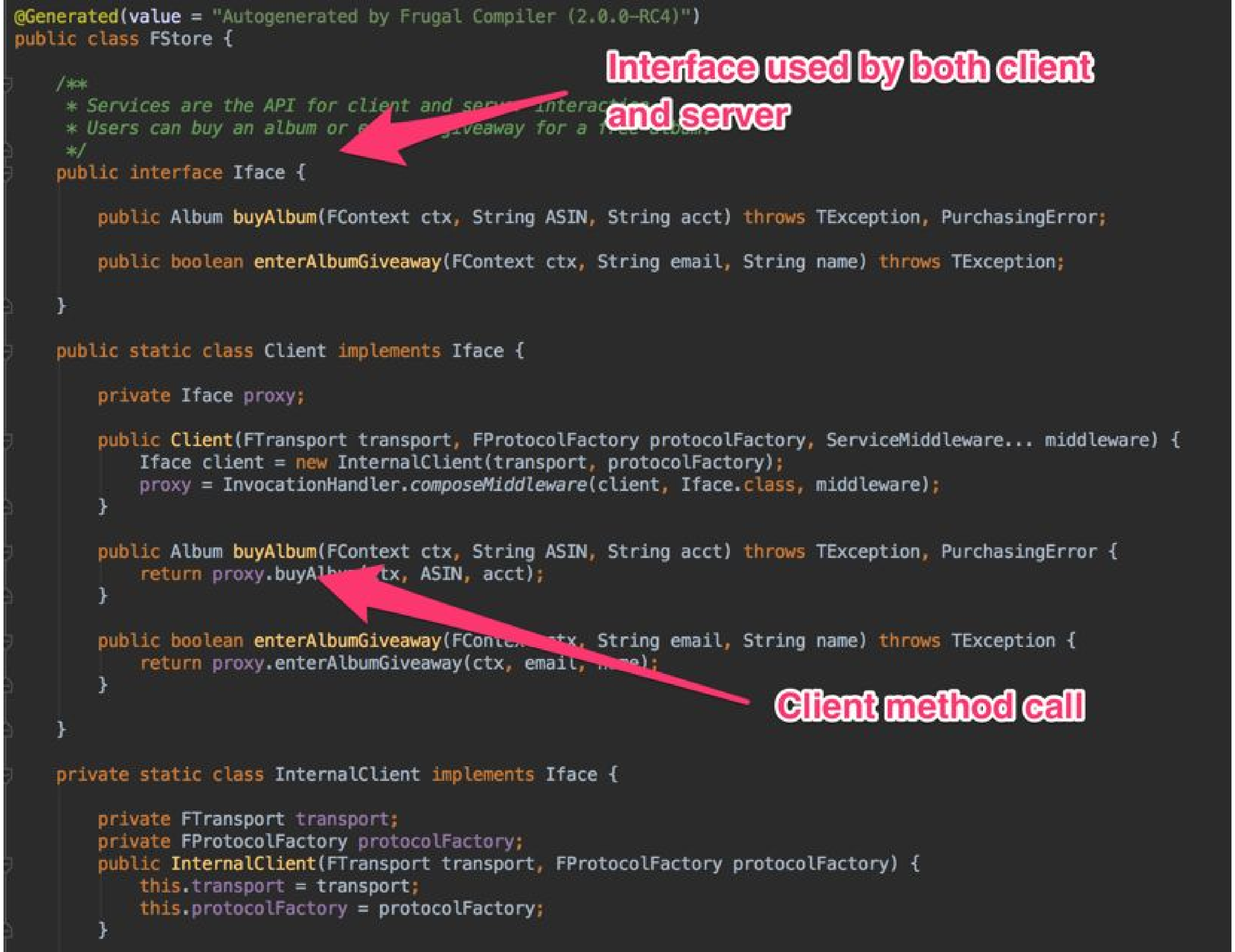
Let’s walk through some generated code to see how this is wired up.

Working off the `Store` example service using the NATS transport/server.



```
/**@
 * Services are the API for client and server interaction.
 * Users can buy an album or enter a giveaway for a free album.
 */
service Store {
    Album buyAlbum( 1: string ASIN, 2: string acct ) throws (1: PurchasingError error)
    bool enterAlbumGiveaway( 1: string email, 2: string name )
}
```

Generated API

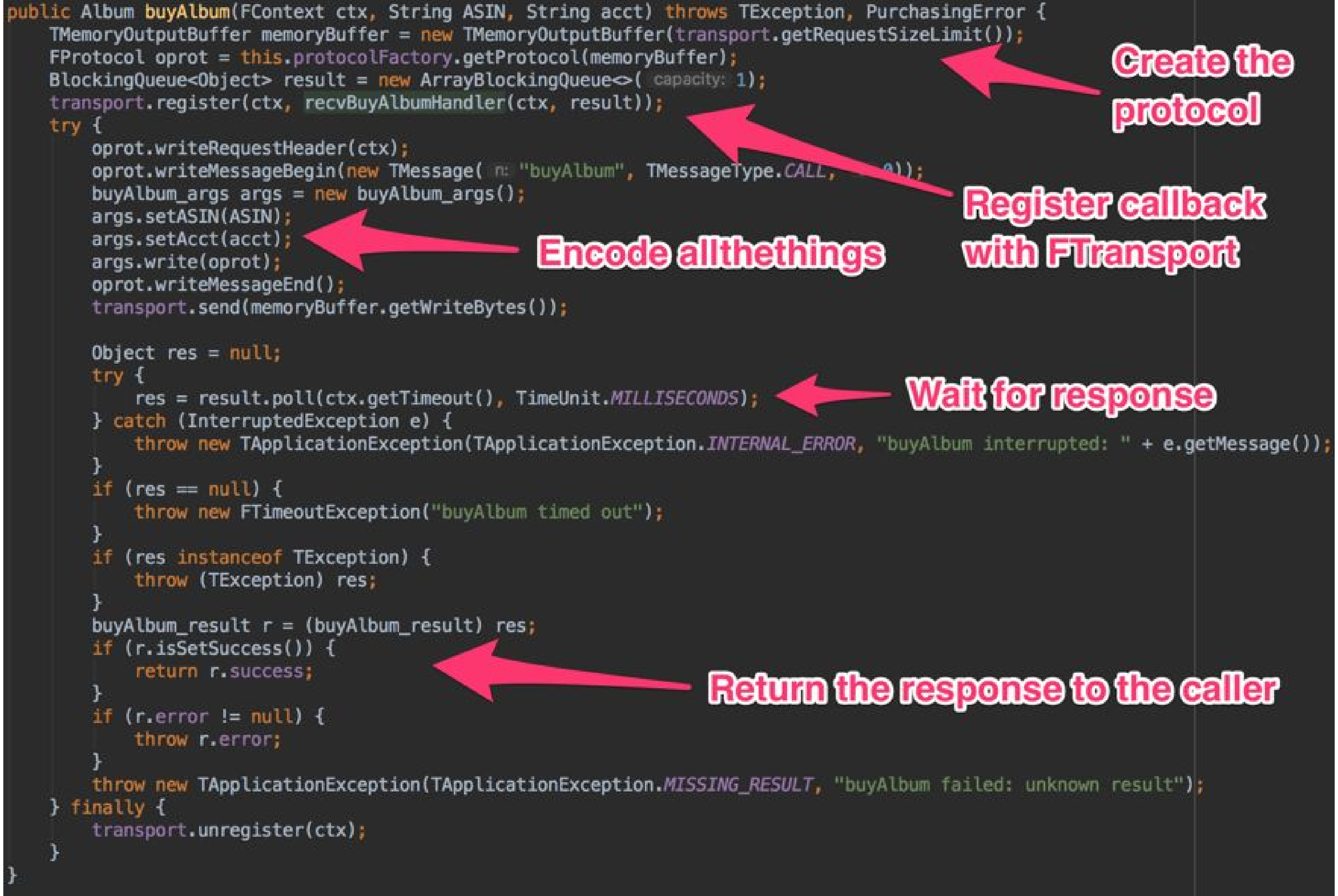


```java
@Generated(value = "Autogenerated by Frugal Compiler (2.0.0-RC4)")
public class FStore {

    /**
     * Services are the API for client and server interaction.
     * Users can buy an album or enter a giveaway for a free album.
     */
    public interface Iface {

        public Album buyAlbum(FContext ctx, String ASIN, String acct) throws TException, PurchasingError;

        public boolean enterAlbumGiveaway(FContext ctx, String email, String name) throws TException;

    }

    public static class Client implements Iface {

        private Iface proxy;

        public Client(FTransport transport, FProtocolFactory protocolFactory, ServiceMiddleware... middleware) {
            Iface client = new InternalClient(transport, protocolFactory);
            proxy = InvocationHandler.composeMiddleware(client, Iface.class, middleware);
        }

        public Album buyAlbum(FContext ctx, String ASIN, String acct) throws TException, PurchasingError {
            return proxy.buyAlbum(ctx, ASIN, acct);
        }

        public boolean enterAlbumGiveaway(FContext ctx, String email, String name) throws TException {
            return proxy.enterAlbumGiveaway(ctx, email, name);
        }

    }

    private static class InternalClient implements Iface {

        private FTransport transport;
        private FProtocolFactory protocolFactory;
        public InternalClient(FTransport transport, FProtocolFactory protocolFactory) {
            this.transport = transport;
            this.protocolFactory = protocolFactory;
        }
```

Interface used by both client and server

Client method call

Generated API



```
public Album buyAlbum(FContext ctx, String ASIN, String acct) throws TException, PurchasingError {
    TMemoryOutputBuffer memoryBuffer = new TMemoryOutputBuffer(transport.getRequestSizeLimit());
    FProtocol oprot = this.protocolFactory.getProtocol(memoryBuffer);
    BlockingQueue<Object> result = new ArrayBlockingQueue<>( capacity: 1);
    transport.register(ctx, recvBuyAlbumHandler(ctx, result));
    try {
        oprot.writeRequestHeader(ctx);
        oprot.writeMessageBegin(new TMessage( n: "buyAlbum", TMessageType.CALL,      0));
        buyAlbum_args args = new buyAlbum_args();
        args.setASIN(ASIN);
        args.setAcct(acct);
        args.write(oprot);
        oprot.writeMessageEnd();
        transport.send(memoryBuffer.getWriteBytes());

        Object res = null;
        try {
            res = result.poll(ctx.getTimeout(), TimeUnit.MILLISECONDS);
        } catch (InterruptedException e) {
            throw new TApplicationException(TApplicationException.INTERNAL_ERROR, "buyAlbum interrupted: " + e.getMessage());
        }
        if (res == null) {
            throw new FTimeoutException("buyAlbum timed out");
        }
        if (res instanceof TException) {
            throw (TException) res;
        }
        buyAlbum_result r = (buyAlbum_result) res;
        if (r.isSetSuccess()) {
            return r.success;
        }
        if (r.error != null) {
            throw r.error;
        }
        throw new TApplicationException(TApplicationException.MISSING_RESULT, "buyAlbum failed: unknown result");
    } finally {
        transport.unregister(ctx);
    }
}
```

Create the protocol

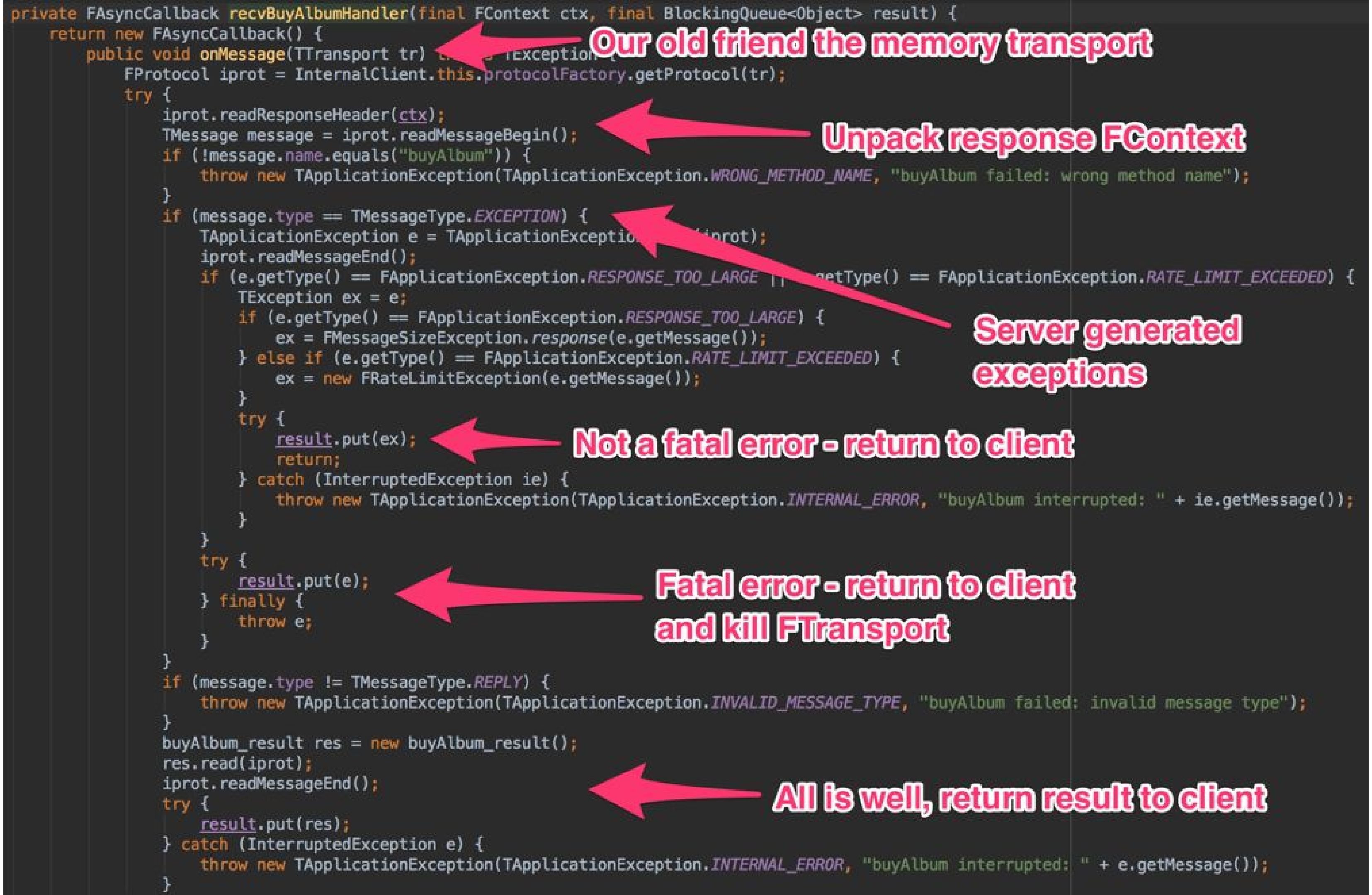
Register callback with FTransport

Encode allthethings

Wait for response

Return the response to the caller

Generated API



```
private FAsyncCallback recvBuyAlbumHandler(final FContext ctx, final BlockingQueue<Object> result) {
    return new FAsyncCallback() {
        public void onMessage(TTransport tr) throws TException {
            FProtocol iprot = InternalClient.this.protocolFactory.getProtocol(tr);
            try {
                iprot.readResponseHeader(ctx);
                TMessage message = iprot.readMessageBegin();
                if (!message.name.equals("buyAlbum")) {
                    throw new TApplicationException(TApplicationException.WRONG_METHOD_NAME, "buyAlbum failed: wrong method name");
                }
                if (message.type == TMessageType.EXCEPTION) {
                    TApplicationException e = TApplicationException.read(iprot);
                    iprot.readMessageEnd();
                    if (e.getType() == FApplicationException.RESPONSE_TOO_LARGE || e.getType() == FApplicationException.RATE_LIMIT_EXCEEDED) {
                        TException ex = e;
                        if (e.getType() == FApplicationException.RESPONSE_TOO_LARGE) {
                            ex = FMessageSizeException.response(e.getMessage());
                        } else if (e.getType() == FApplicationException.RATE_LIMIT_EXCEEDED) {
                            ex = new FRateLimitException(e.getMessage());
                        }
                        try {
                            result.put(ex);
                            return;
                        } catch (InterruptedException ie) {
                            throw new TApplicationException(TApplicationException.INTERNAL_ERROR, "buyAlbum interrupted: " + ie.getMessage());
                        }
                    }
                    try {
                        result.put(e);
                    } finally {
                        throw e;
                    }
                }
                if (message.type != TMessageType.REPLY) {
                    throw new TApplicationException(TApplicationException.INVALID_MESSAGE_TYPE, "buyAlbum failed: invalid message type");
                }
                buyAlbum_result res = new buyAlbum_result();
                res.read(iprot);
                iprot.readMessageEnd();
                try {
                    result.put(res);
                } catch (InterruptedException e) {
                    throw new TApplicationException(TApplicationException.INTERNAL_ERROR, "buyAlbum interrupted: " + e.getMessage());
                }
```

Our old friend the memory transport

Unpack response FContext

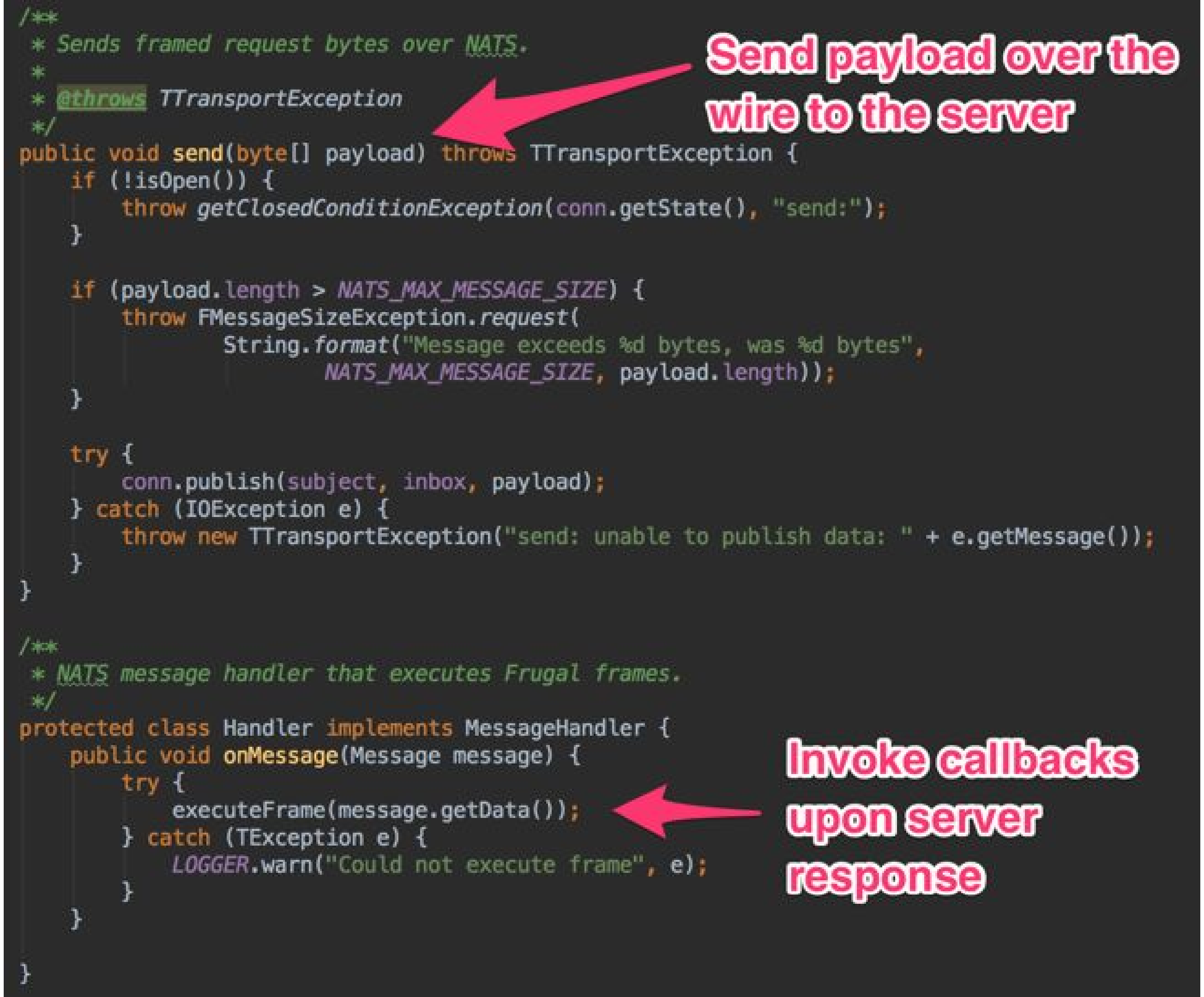
Server generated exceptions

Not a fatal error - return to client

Fatal error - return to client and kill FTransport

All is well, return result to client

FNatsTransport



```java
/**
 * Sends framed request bytes over NATS.
 *
 * @throws TTransportException
 */
public void send(byte[] payload) throws TTransportException {
    if (!isOpen()) {
        throw getClosedConditionException(conn.getState(), "send:");
    }

    if (payload.length > NATS_MAX_MESSAGE_SIZE) {
        throw FMessageSizeException.request(
                String.format("Message exceeds %d bytes, was %d bytes",
                        NATS_MAX_MESSAGE_SIZE, payload.length));
    }

    try {
        conn.publish(subject, inbox, payload);
    } catch (IOException e) {
        throw new TTransportException("send: unable to publish data: " + e.getMessage());
    }
}

/**
 * NATS message handler that executes Frugal frames.
 */
protected class Handler implements MessageHandler {
    public void onMessage(Message message) {
        try {
            executeFrame(message.getData());
        } catch (TException e) {
            LOGGER.warn("Could not execute frame", e);
        }
    }

}
```

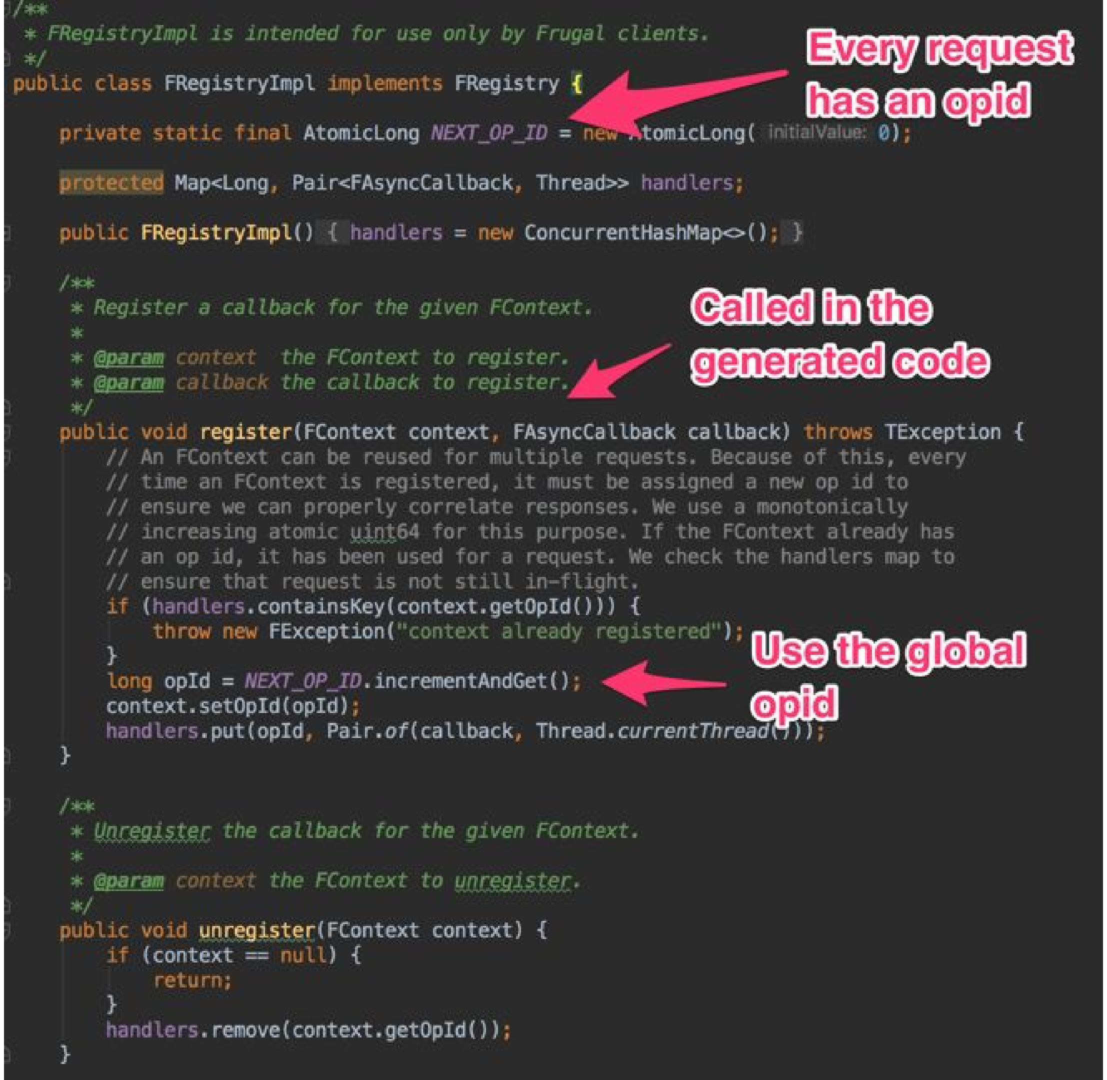
Send payload over the wire to the server

Invoke callbacks upon server response

BONUS TIME!

FRegistry

FRegistryImpl



```java
/**
 * FRegistryImpl is intended for use only by Frugal clients.
 */
public class FRegistryImpl implements FRegistry {

    private static final AtomicLong NEXT_OP_ID = new AtomicLong( initialValue: 0);

    protected Map<Long, Pair<FAsyncCallback, Thread>> handlers;

    public FRegistryImpl() { handlers = new ConcurrentHashMap<>(); }

    /**
     * Register a callback for the given FContext.
     *
     * @param context  the FContext to register.
     * @param callback the callback to register.
     */
    public void register(FContext context, FAsyncCallback callback) throws TException {
        // An FContext can be reused for multiple requests. Because of this, every
        // time an FContext is registered, it must be assigned a new op id to
        // ensure we can properly correlate responses. We use a monotonically
        // increasing atomic uint64 for this purpose. If the FContext already has
        // an op id, it has been used for a request. We check the handlers map to
        // ensure that request is not still in-flight.
        if (handlers.containsKey(context.getOpId())) {
            throw new FException("context already registered");
        }
        long opId = NEXT_OP_ID.incrementAndGet();
        context.setOpId(opId);
        handlers.put(opId, Pair.of(callback, Thread.currentThread()));
    }

    /**
     * Unregister the callback for the given FContext.
     *
     * @param context the FContext to unregister.
     */
    public void unregister(FContext context) {
        if (context == null) {
            return;
        }
        handlers.remove(context.getOpId());
    }
```

Every request has an opid

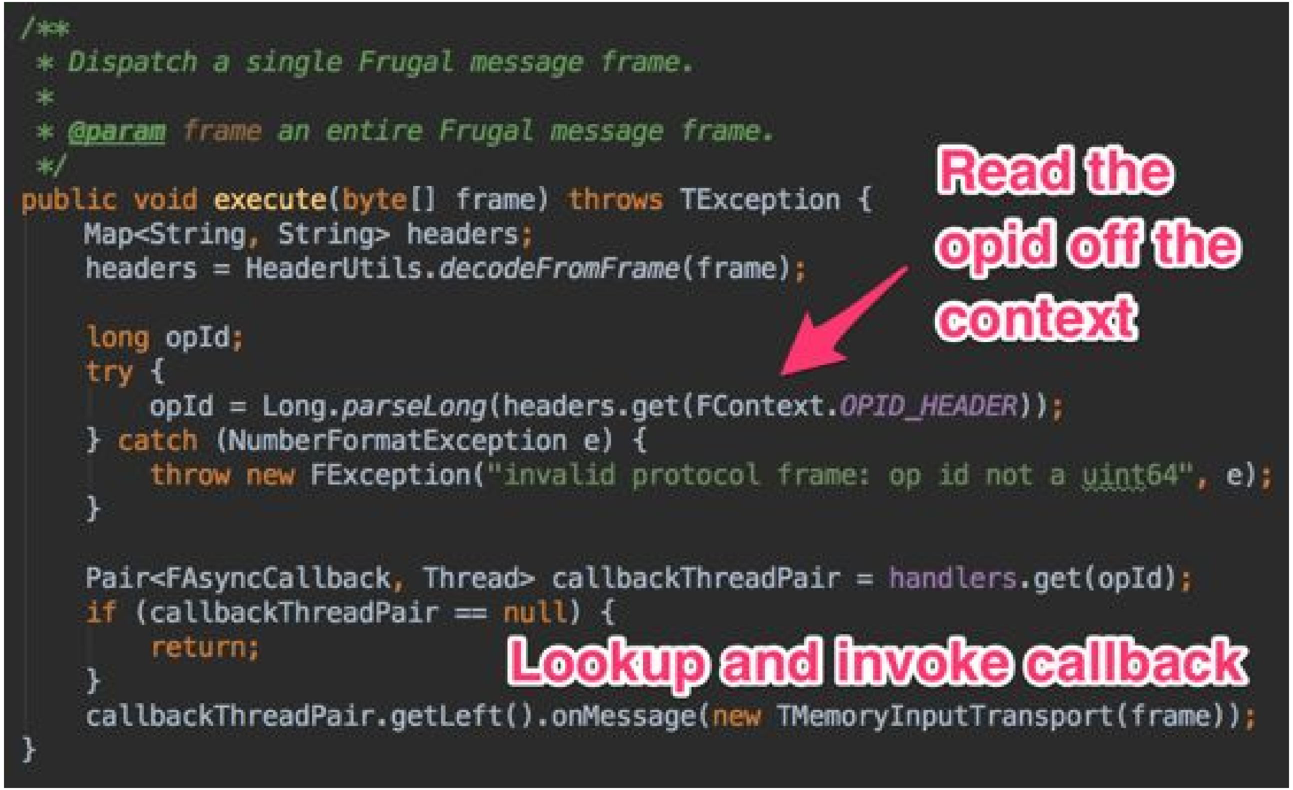
Called in the generated code

Use the global opid

BONUS TIME!

FRegistry

FRegistryImpl



```
/**
 * Dispatch a single Frugal message frame.
 *
 * @param frame an entire Frugal message frame.
 */
public void execute(byte[] frame) throws TException {
    Map<String, String> headers;
    headers = HeaderUtils.decodeFromFrame(frame);

    long opId;
    try {
        opId = Long.parseLong(headers.get(FContext.OPID_HEADER));
    } catch (NumberFormatException e) {
        throw new FException("invalid protocol frame: op id not a uint64", e);
    }

    Pair<FAsyncCallback, Thread> callbackThreadPair = handlers.get(opId);
    if (callbackThreadPair == null) {
        return;
    }
    callbackThreadPair.getLeft().onMessage(new TMemoryInputTransport(frame));
}
```

Read the opid off the context

Lookup and invoke callback

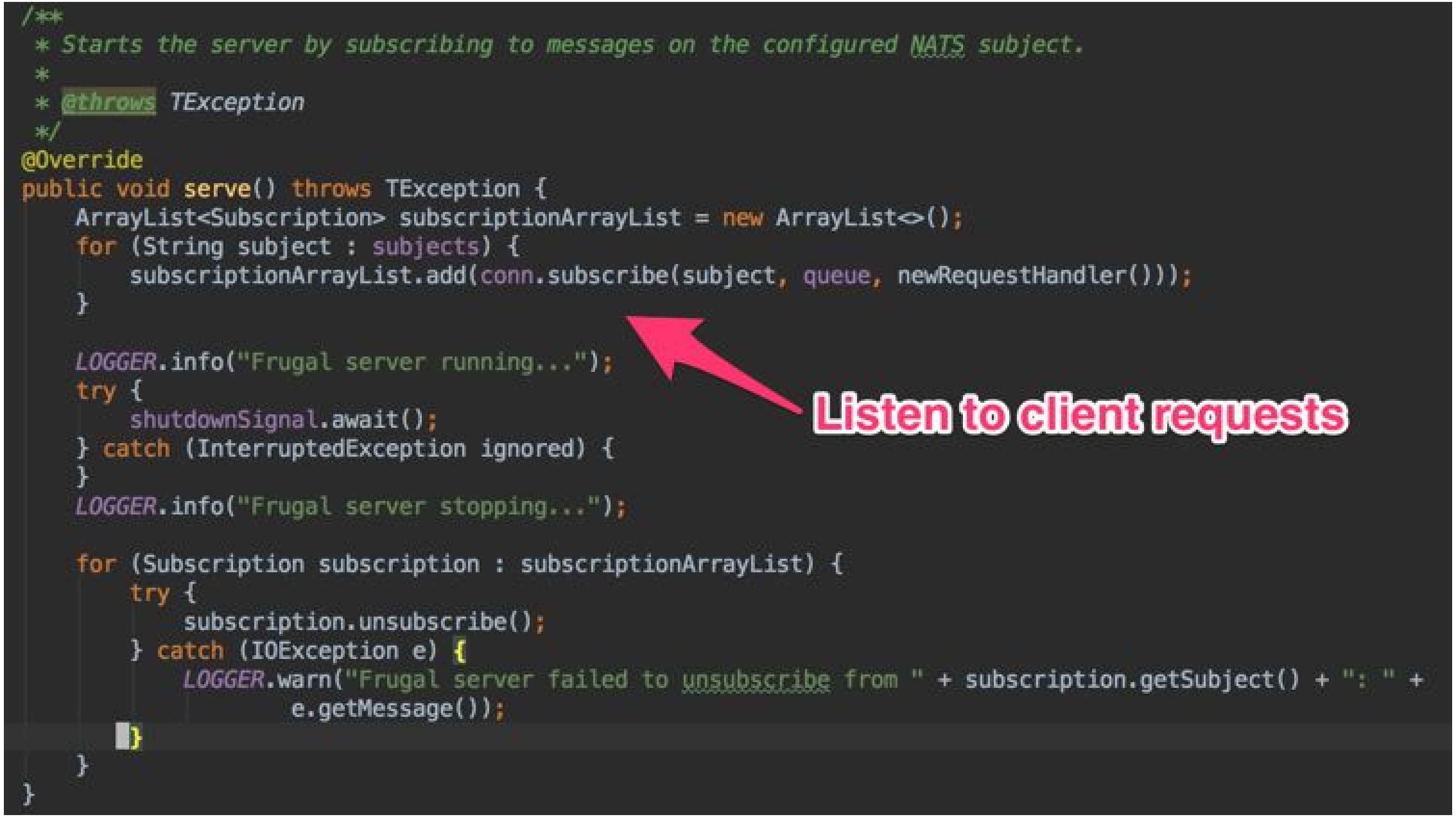
Server

There are three basic components to the server

1. Handler implementation - the server must actually do things in response to user
2. Processor - generated by the frugal compiler in combination with a base processor
    - deserialized the wire-level bytes into memory objects
    - hands the memory object to the handler
    - serializes the response
3. FServer - handles accepting requests off the wire and handing them to the processor, sends responses from the processor back to the client

Back to code!

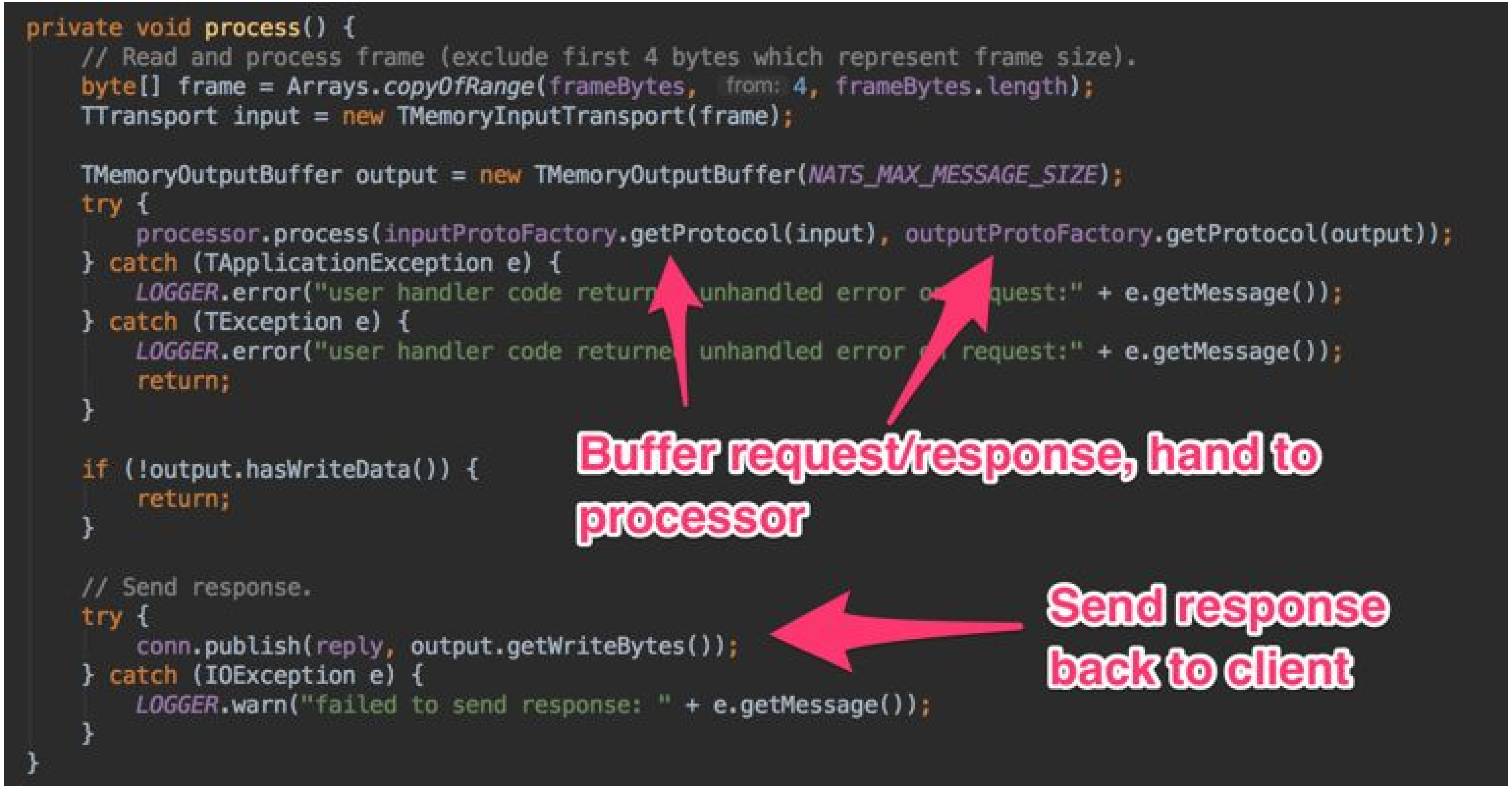
FNatsServer



```java
/**
 * Starts the server by subscribing to messages on the configured NATS subject.
 *
 * @throws TException
 */
@Override
public void serve() throws TException {
    ArrayList<Subscription> subscriptionArrayList = new ArrayList<>();
    for (String subject : subjects) {
        subscriptionArrayList.add(conn.subscribe(subject, queue, newRequestHandler()));
    }

    LOGGER.info("Frugal server running...");
    try {
        shutdownSignal.await();
    } catch (InterruptedException ignored) {
    }
    LOGGER.info("Frugal server stopping...");

    for (Subscription subscription : subscriptionArrayList) {
        try {
            subscription.unsubscribe();
        } catch (IOException e) {
            LOGGER.warn("Frugal server failed to unsubscribe from " + subscription.getSubject() + ": " +
                    e.getMessage());
        }
    }
}
```

Listen to client requests

FNatsServer

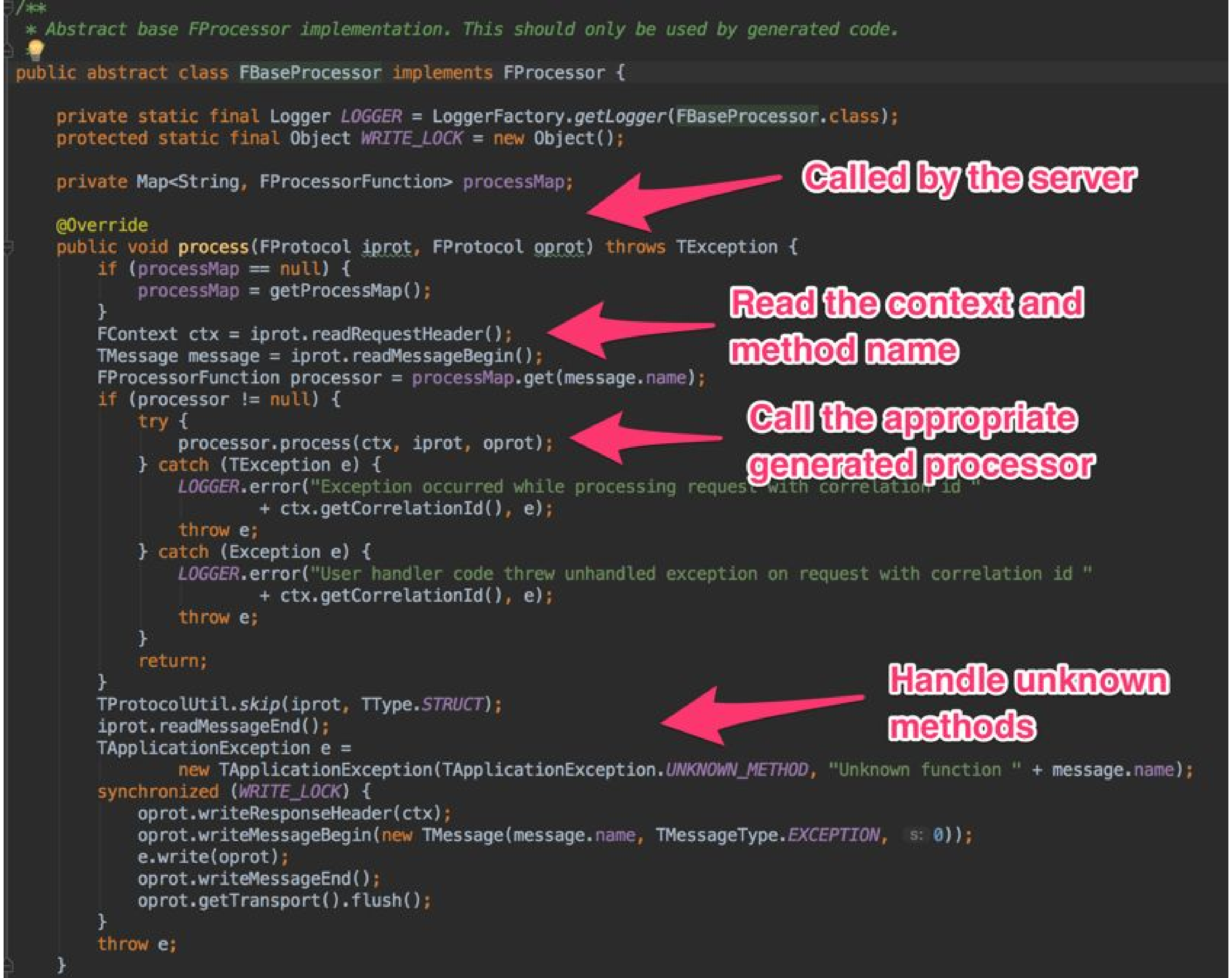


```java
private void process() {
    // Read and process frame (exclude first 4 bytes which represent frame size).
    byte[] frame = Arrays.copyOfRange(frameBytes, from: 4, frameBytes.length);
    TTransport input = new TMemoryInputTransport(frame);

    TMemoryOutputBuffer output = new TMemoryOutputBuffer(NATS_MAX_MESSAGE_SIZE);
    try {
        processor.process(inputProtoFactory.getProtocol(input), outputProtoFactory.getProtocol(output));
    } catch (TApplicationException e) {
        LOGGER.error("user handler code returned unhandled error on request:" + e.getMessage());
    } catch (TException e) {
        LOGGER.error("user handler code returned unhandled error on request:" + e.getMessage());
        return;
    }

    if (!output.hasWriteData()) {
        return;
    }

    // Send response.
    try {
        conn.publish(reply, output.getWriteBytes());
    } catch (IOException e) {
        LOGGER.warn("failed to send response: " + e.getMessage());
    }
}
```

Buffer request/response, hand to processor

Send response back to client

FBaseProcessor



```java
/**
 * Abstract base FProcessor implementation. This should only be used by generated code.

public abstract class FBaseProcessor implements FProcessor {

    private static final Logger LOGGER = LoggerFactory.getLogger(FBaseProcessor.class);
    protected static final Object WRITE_LOCK = new Object();

    private Map<String, FProcessorFunction> processMap;

    @Override
    public void process(FProtocol iprot, FProtocol oprot) throws TException {
        if (processMap == null) {
            processMap = getProcessMap();
        }
        FContext ctx = iprot.readRequestHeader();
        TMessage message = iprot.readMessageBegin();
        FProcessorFunction processor = processMap.get(message.name);
        if (processor != null) {
            try {
                processor.process(ctx, iprot, oprot);
            } catch (TException e) {
                LOGGER.error("Exception occurred while processing request with correlation id "
                        + ctx.getCorrelationId(), e);
                throw e;
            } catch (Exception e) {
                LOGGER.error("User handler code threw unhandled exception on request with correlation id "
                        + ctx.getCorrelationId(), e);
                throw e;
            }
            return;
        }
        TProtocolUtil.skip(iprot, TType.STRUCT);
        iprot.readMessageEnd();
        TApplicationException e =
                new TApplicationException(TApplicationException.UNKNOWN_METHOD, "Unknown function " + message.name);
        synchronized (WRITE_LOCK) {
            oprot.writeResponseHeader(ctx);
            oprot.writeMessageBegin(new TMessage(message.name, TMessageType.EXCEPTION,  s: 0));
            e.write(oprot);
            oprot.writeMessageEnd();
            oprot.getTransport().flush();
        }
        throw e;
    }
```

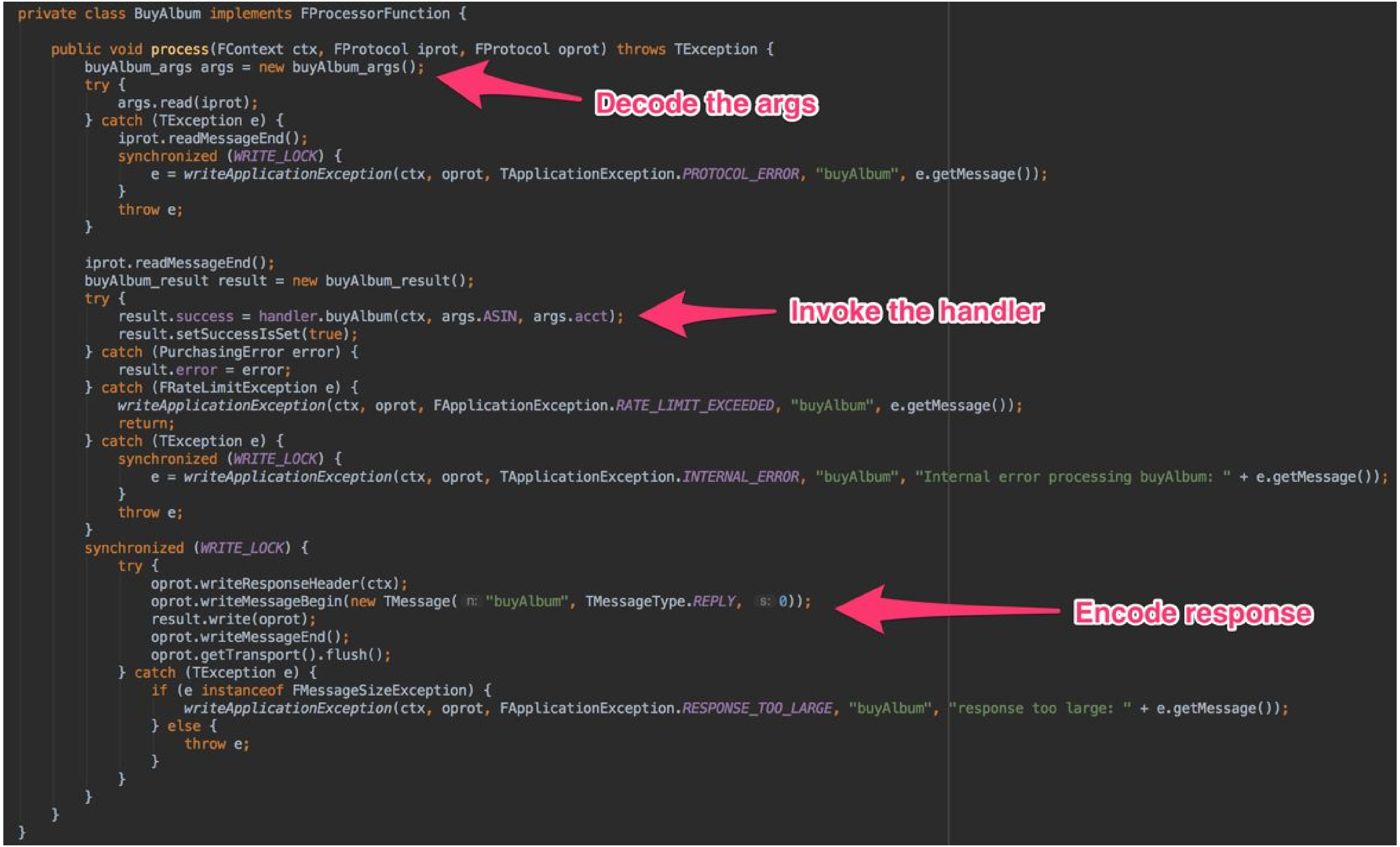
Called by the server

Read the context and method name

Call the appropriate generated processor

Handle unknown methods

Generated processor



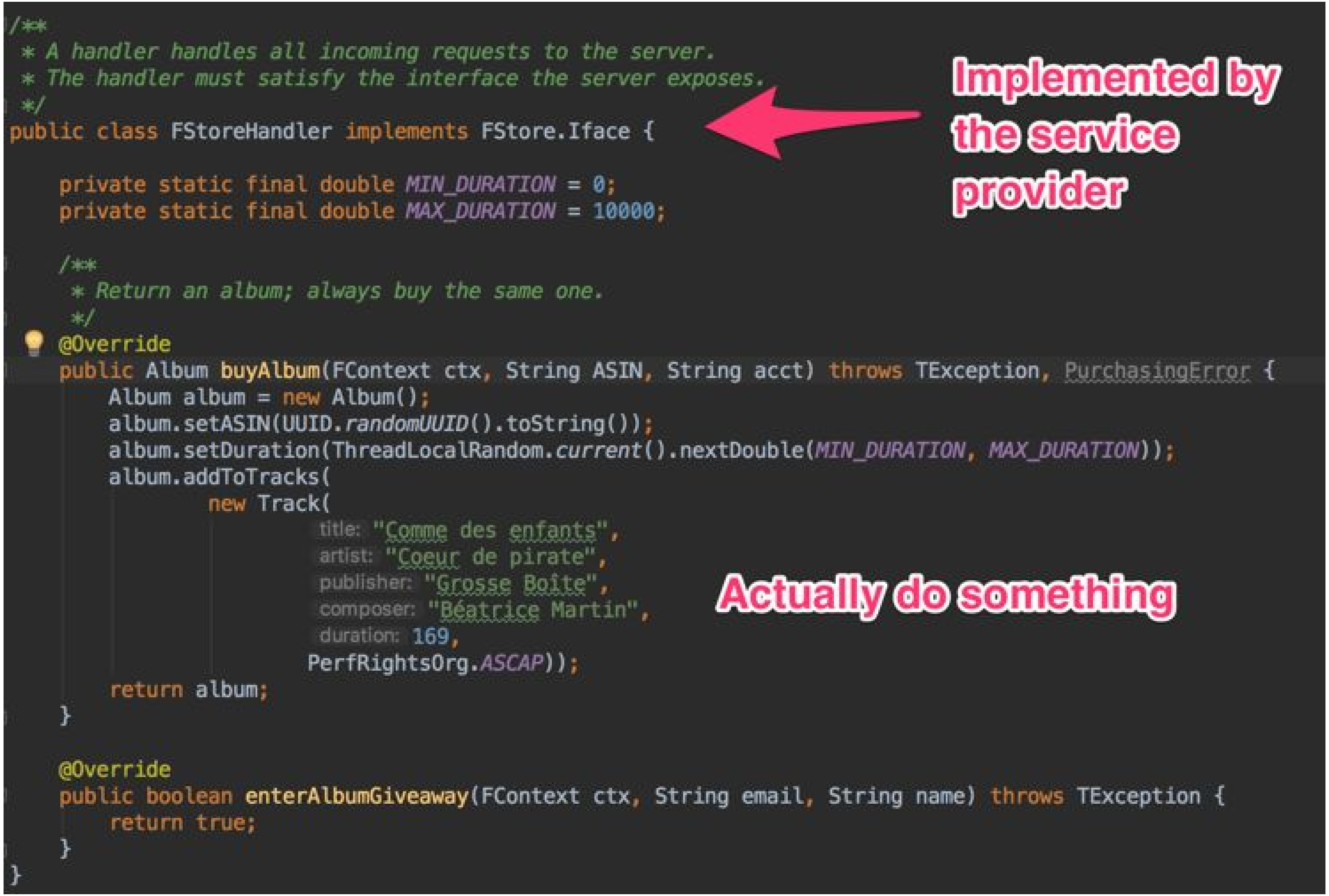
```
private class BuyAlbum implements FProcessorFunction {

    public void process(FContext ctx, FProtocol iprot, FProtocol oprot) throws TException {
        buyAlbum_args args = new buyAlbum_args();
        try {
            args.read(iprot);
        } catch (TException e) {
            iprot.readMessageEnd();
            synchronized (WRITE_LOCK) {
                e = writeApplicationException(ctx, oprot, TApplicationException.PROTOCOL_ERROR, "buyAlbum", e.getMessage());
            }
            throw e;
        }

        iprot.readMessageEnd();
        buyAlbum_result result = new buyAlbum_result();
        try {
            result.success = handler.buyAlbum(ctx, args.ASIN, args.acct);
            result.setSuccessIsSet(true);
        } catch (PurchasingError error) {
            result.error = error;
        } catch (FRateLimitException e) {
            writeApplicationException(ctx, oprot, FApplicationException.RATE_LIMIT_EXCEEDED, "buyAlbum", e.getMessage());
            return;
        } catch (TException e) {
            synchronized (WRITE_LOCK) {
                e = writeApplicationException(ctx, oprot, TApplicationException.INTERNAL_ERROR, "buyAlbum", "Internal error processing buyAlbum: " + e.getMessage());
            }
            throw e;
        }
        synchronized (WRITE_LOCK) {
            try {
                oprot.writeResponseHeader(ctx);
                oprot.writeMessageBegin(new TMessage( n: "buyAlbum", TMessageType.REPLY,  s: 0));
                result.write(oprot);
                oprot.writeMessageEnd();
                oprot.getTransport().flush();
            } catch (TException e) {
                if (e instanceof FMessageSizeException) {
                    writeApplicationException(ctx, oprot, FApplicationException.RESPONSE_TOO_LARGE, "buyAlbum", "response too large: " + e.getMessage());
                } else {
                    throw e;
                }
            }
        }
    }
}
```

Decode the args

Invoke the handler

Encode response

Service handler

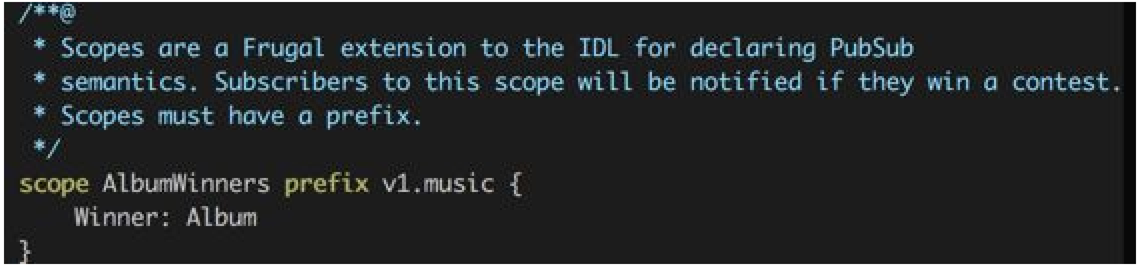


```java
/**
 * A handler handles all incoming requests to the server.
 * The handler must satisfy the interface the server exposes.
 */
public class FStoreHandler implements FStore.Iface {

    private static final double MIN_DURATION = 0;
    private static final double MAX_DURATION = 10000;

    /**
     * Return an album; always buy the same one.
     */
    @Override
    public Album buyAlbum(FContext ctx, String ASIN, String acct) throws TException, PurchasingError {
        Album album = new Album();
        album.setASIN(UUID.randomUUID().toString());
        album.setDuration(ThreadLocalRandom.current().nextDouble(MIN_DURATION, MAX_DURATION));
        album.addToTracks(
                new Track(
                        title: "Comme des enfants",
                        artist: "Coeur de pirate",
                        publisher: "Grosse Boîte",
                        composer: "Béatrice Martin",
                        duration: 169,
                        PerfRightsOrg.ASCAP));
        return album;
    }

    @Override
    public boolean enterAlbumGiveaway(FContext ctx, String email, String name) throws TException {
        return true;
    }
}
```

Implemented by the service provider

Actually do something

Working off the `AlbumWinners` example scope using the NATS transports.



```
/**@
 * Scopes are a Frugal extension to the IDL for declaring PubSub
 * semantics. Subscribers to this scope will be notified if they win a contest.
 * Scopes must have a prefix.
 */
scope AlbumWinners prefix v1.music {
    Winner: Album
}
```

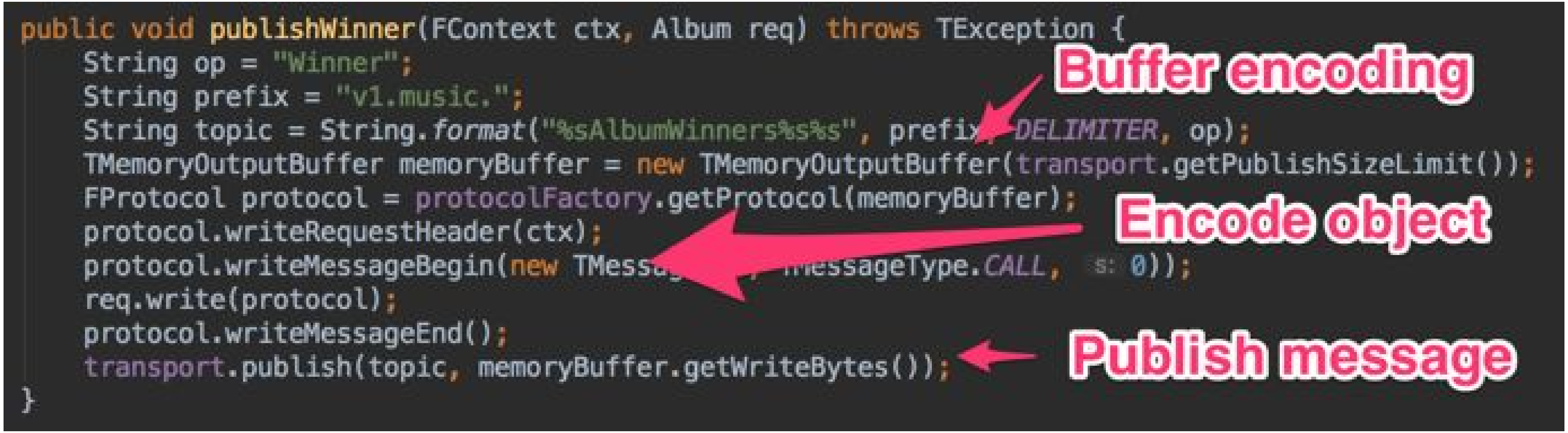
Scope Publisher

There are two basic components to the publisher

1. Publish API - generated by the frugal compiler -
    - generates the scope topic to be used by the transport
    - serializes the memory objects and hands them to the FPublisherTransport
2. FPublisherTransport - interface - publishes serialized data

Code again? Ok, I guess...

Generated publisher



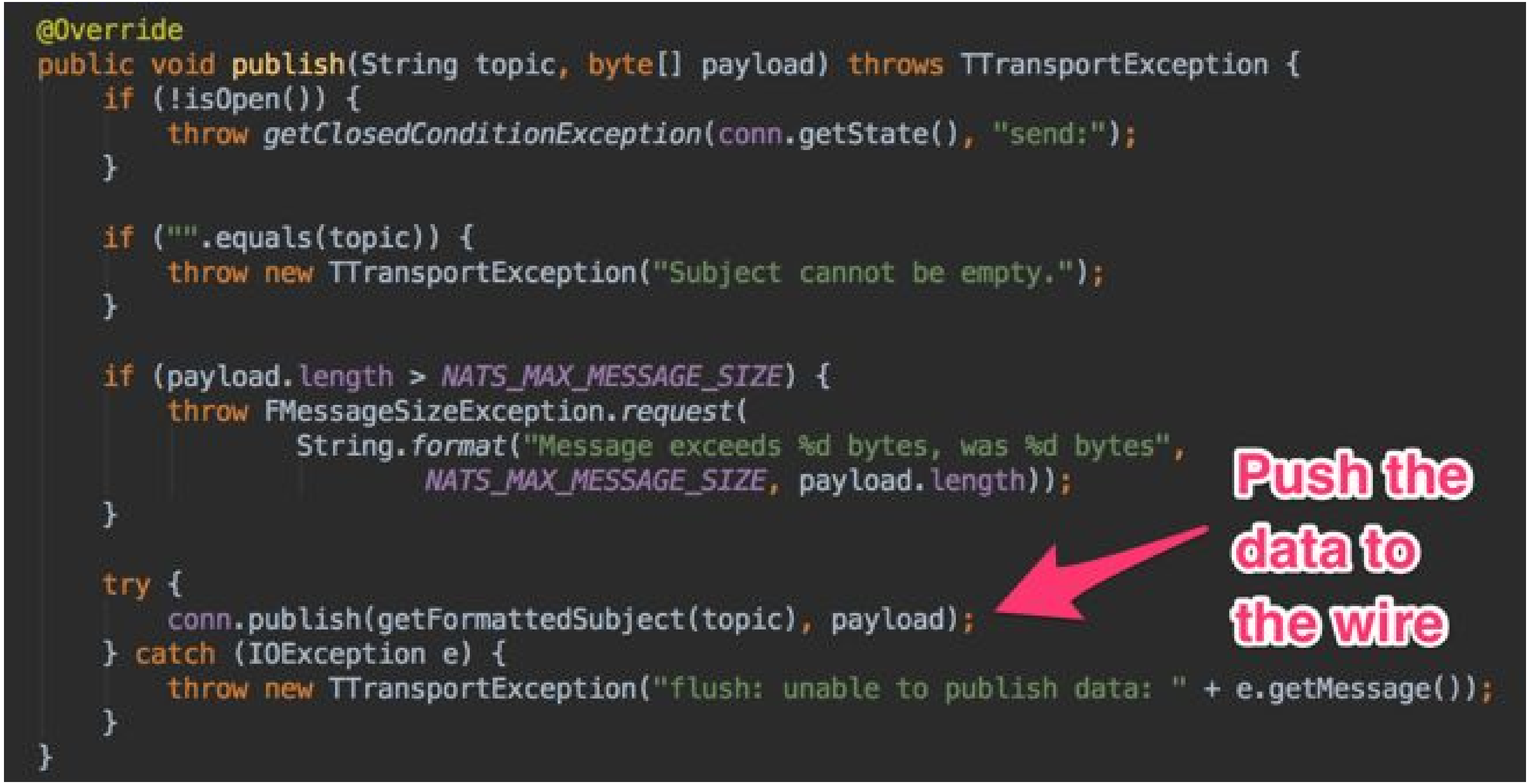
```
public void publishWinner(FContext ctx, Album req) throws TException {
    String op = "Winner";
    String prefix = "v1.music.";
    String topic = String.format("%sAlbumWinners%s%s", prefix, DELIMITER, op);
    TMemoryOutputBuffer memoryBuffer = new TMemoryOutputBuffer(transport.getPublishSizeLimit());
    FProtocol protocol = protocolFactory.getProtocol(memoryBuffer);
    protocol.writeRequestHeader(ctx);
    protocol.writeMessageBegin(new TMessa[illegible], TMessageType.CALL, s: 0));
    req.write(protocol);
    protocol.writeMessageEnd();
    transport.publish(topic, memoryBuffer.getWriteBytes());
}
```

Buffer encoding

Encode object

Publish message

FNatsPublisher



```java
@Override
public void publish(String topic, byte[] payload) throws TTransportException {
    if (!isOpen()) {
        throw getClosedConditionException(conn.getState(), "send:");
    }

    if ("".equals(topic)) {
        throw new TTransportException("Subject cannot be empty.");
    }

    if (payload.length > NATS_MAX_MESSAGE_SIZE) {
        throw FMessageSizeException.request(
                String.format("Message exceeds %d bytes, was %d bytes",
                        NATS_MAX_MESSAGE_SIZE, payload.length));
    }

    try {
        conn.publish(getFormattedSubject(topic), payload);
    } catch (IOException e) {
        throw new TTransportException("flush: unable to publish data: " + e.getMessage());
    }
}
```

Push the data to the wire

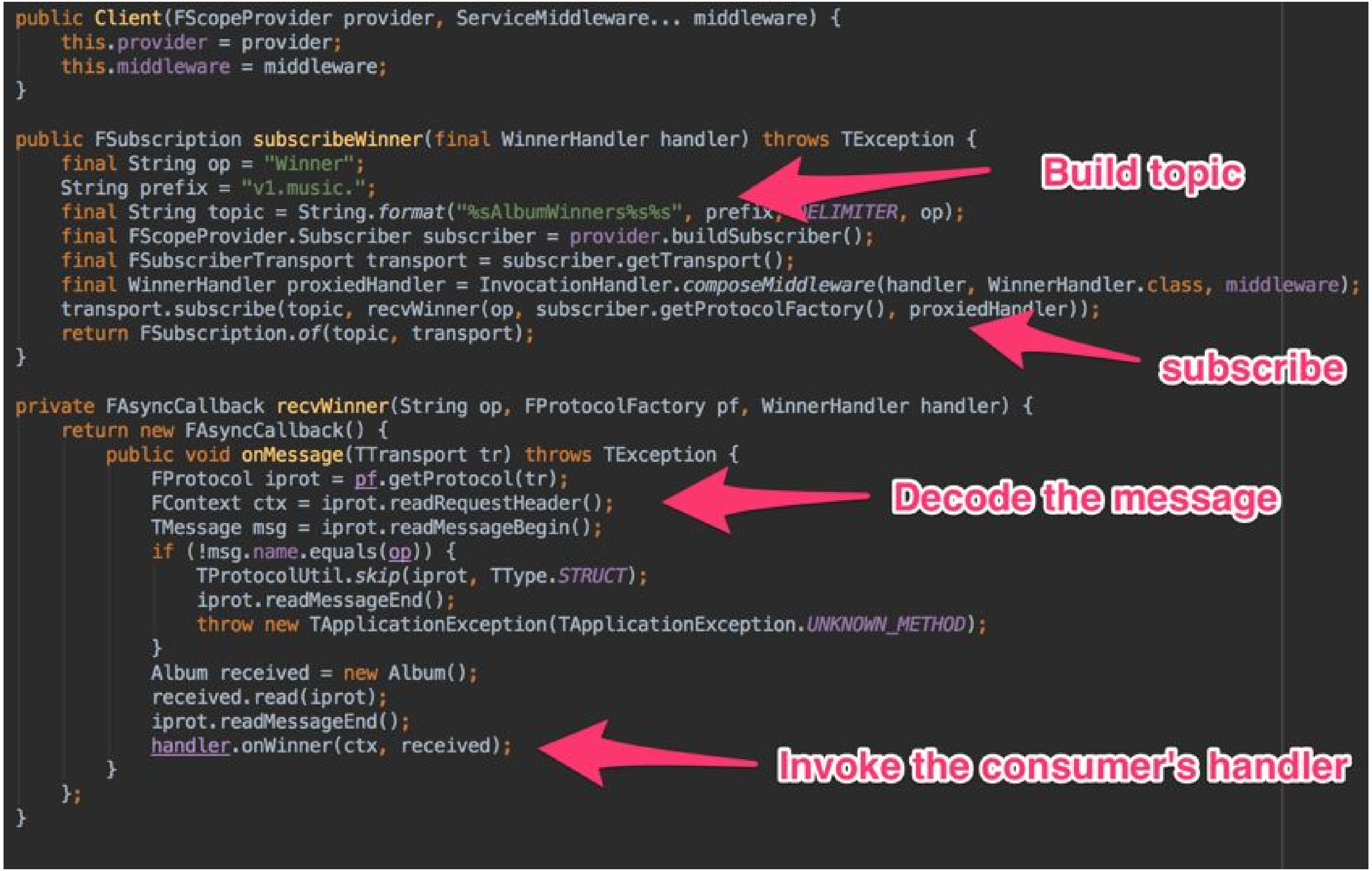
Scope Subscriber

There are two basic components to the subscriber

1. Subscriber API - generated by the frugal compiler -
    - generates the scope topic to be used by the transport
    - creates a new FSubscriberTransport instance listening on the given topic
    - deserializes data coming from the FSubscriberTransport
    - invokes the consumer defined handler on resulting the memory object
2. FSubscriberTransport - interface - subscribes to serialized data

Code? Seriously? This is just getting old.

Generated subscriber



```
public Client(FScopeProvider provider, ServiceMiddleware... middleware) {
    this.provider = provider;
    this.middleware = middleware;
}

public FSubscription subscribeWinner(final WinnerHandler handler) throws TException {
    final String op = "Winner";
    String prefix = "v1.music.";
    final String topic = String.format("%sAlbumWinners%s%s", prefix, DELIMITER, op);
    final FScopeProvider.Subscriber subscriber = provider.buildSubscriber();
    final FSubscriberTransport transport = subscriber.getTransport();
    final WinnerHandler proxiedHandler = InvocationHandler.composeMiddleware(handler, WinnerHandler.class, middleware);
    transport.subscribe(topic, recvWinner(op, subscriber.getProtocolFactory(), proxiedHandler));
    return FSubscription.of(topic, transport);
}

private FAsyncCallback recvWinner(String op, FProtocolFactory pf, WinnerHandler handler) {
    return new FAsyncCallback() {
        public void onMessage(TTransport tr) throws TException {
            FProtocol iprot = pf.getProtocol(tr);
            FContext ctx = iprot.readRequestHeader();
            TMessage msg = iprot.readMessageBegin();
            if (!msg.name.equals(op)) {
                TProtocolUtil.skip(iprot, TType.STRUCT);
                iprot.readMessageEnd();
                throw new TApplicationException(TApplicationException.UNKNOWN_METHOD);
            }
            Album received = new Album();
            received.read(iprot);
            iprot.readMessageEnd();
            handler.onWinner(ctx, received);
        }
    };
}
```

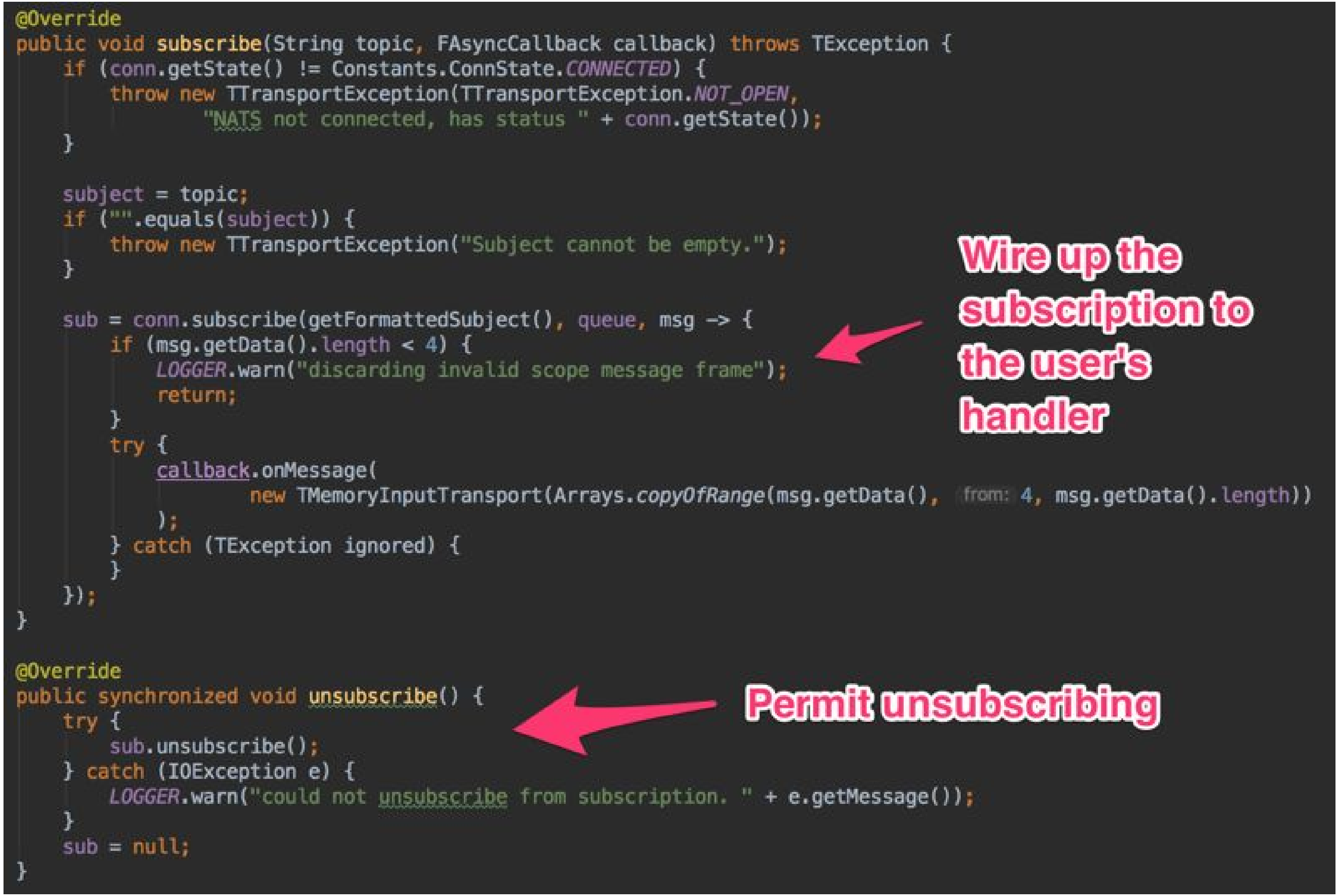
Build topic

subscribe

Decode the message

Invoke the consumer's handler

FNatsSubscriber



```java
@Override
public void subscribe(String topic, FAsyncCallback callback) throws TException {
    if (conn.getState() != Constants.ConnState.CONNECTED) {
        throw new TTransportException(TTransportException.NOT_OPEN,
                "NATS not connected, has status " + conn.getState());
    }

    subject = topic;
    if ("".equals(subject)) {
        throw new TTransportException("Subject cannot be empty.");
    }

    sub = conn.subscribe(getFormattedSubject(), queue, msg -> {
        if (msg.getData().length < 4) {
            LOGGER.warn("discarding invalid scope message frame");
            return;
        }
        try {
            callback.onMessage(
                    new TMemoryInputTransport(Arrays.copyOfRange(msg.getData(),  from: 4, msg.getData().length))
            );
        } catch (TException ignored) {
        }
    });
}

@Override
public synchronized void unsubscribe() {
    try {
        sub.unsubscribe();
    } catch (IOException e) {
        LOGGER.warn("could not unsubscribe from subscription. " + e.getMessage());
    }
    sub = null;
}
```

Wire up the subscription to the user's handler

Permit unsubscribing