BIY - Build It Yourself:

A simple multithreaded REST server using Qt as a powerful building block

Sergio Borghese
NetResults Srl

Who are we...

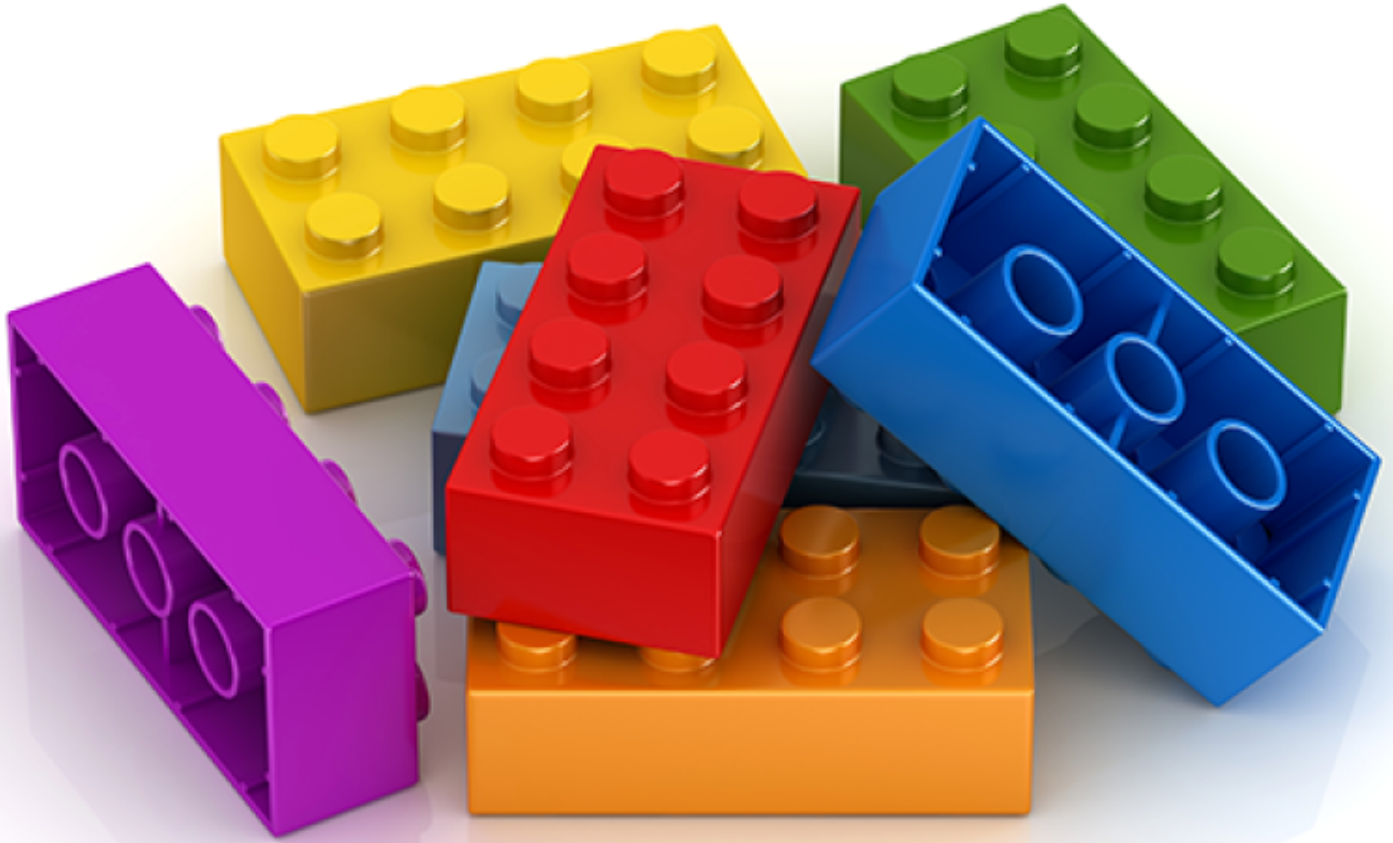
- Established in 2006; University of Pisa Spin-off
  - Research Group of Telecommunication Engineering
- Developing software for Telecommunication Industry

NetResults
Building the digital society

KALLIOPE PBX

NDK VoIPDevelopmentKit

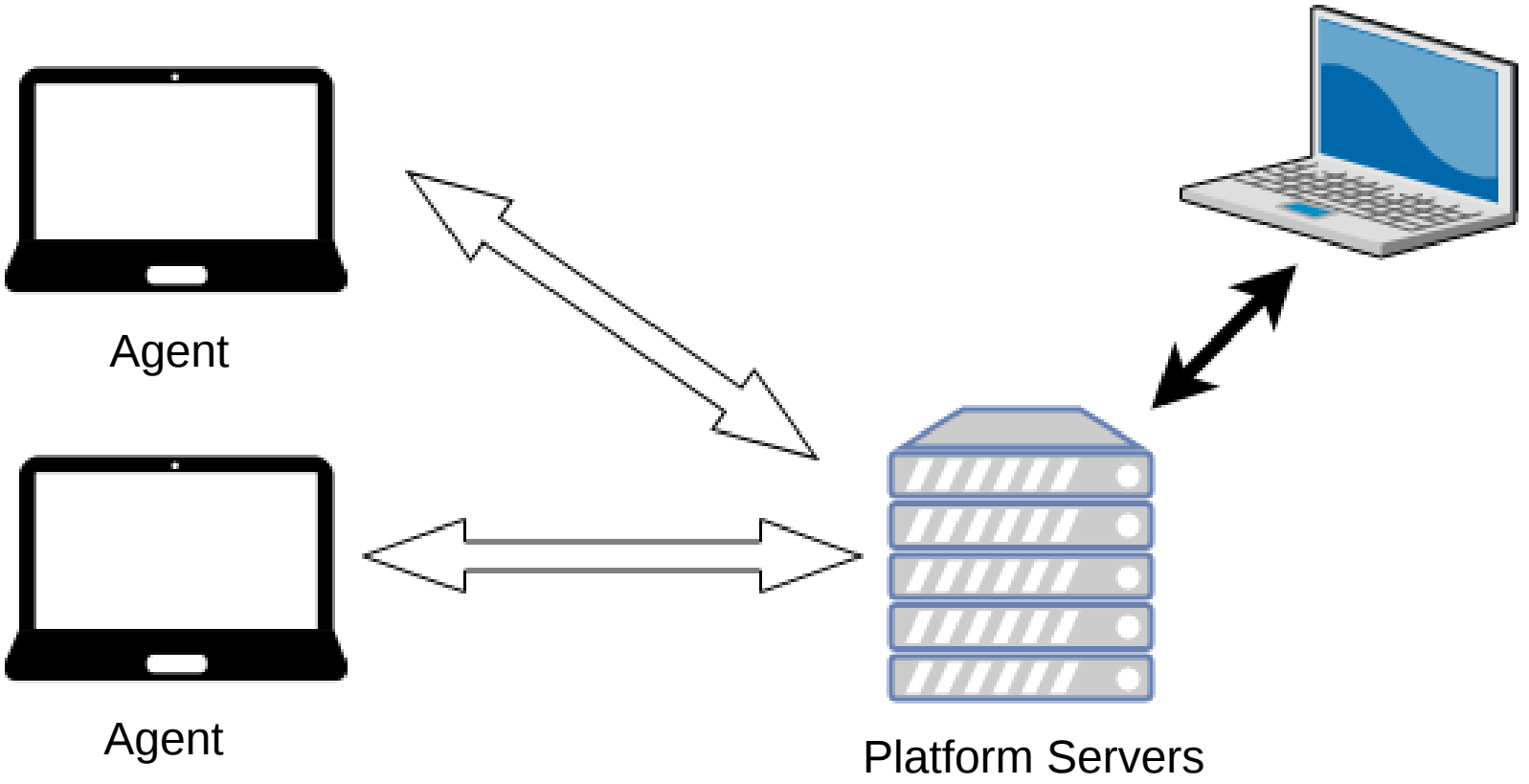
Let's play with Qt



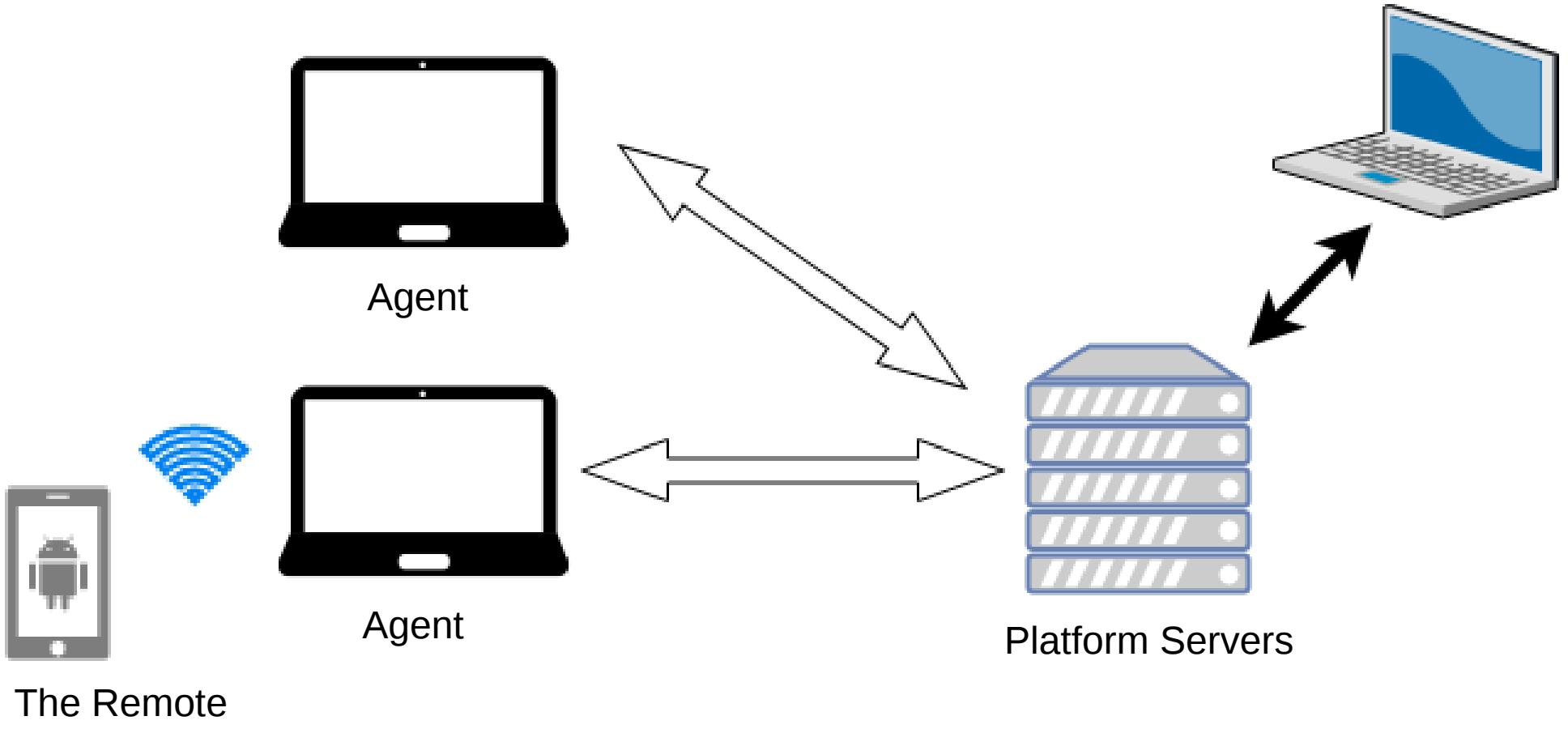
Qt day

Florence, 24th May 2018

Test Platform Architecture (1)



Test Platform Architecture (1)

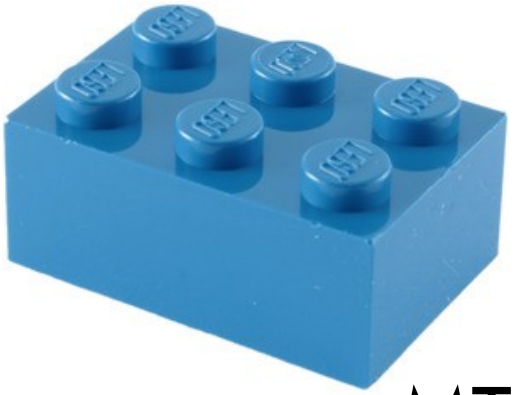


Test Platform Architecture (2)

- Agent and services are **Qt (4.8/5.6)** based
  - ~10 Qt based backend services
- Agents are controlled remotely from the platform
- New Use Case: user installs the agent and needs to command it locally
  - We need a remote controller!
- Add a REST API layer on Agent and servers
  - Agent can directly manage the requests or forward them

The Building Blocks (1)

REST Layer



MT Server

Data Slicer

ThreadPool

...and everything fits together

Qt day

Florence, 24th May 2018

NrThreadPool (1)

- Runs **K** QObject(s) using **N(+1)** Qthread(s)
- Object to Thread allocation policy
  - RoundRobin
  - MinJobs
- Optional Watchdog thread (**the +1**) to try respawning locked threads

```
thPtr->quit();
if (!thPtr->wait(maxTimeoutMsec) )
{
  thPtr→terminate();
  addAnotherThreadToPool();
}
```

NrThreadPool (2)

```
class NRThreadPool : public QObject
{
    Q_OBJECT

private:
    QVector<QThread* > m_v;
    ThreadAssignmentPolicy m_threadUsagePolicy;
    QMutex mux;
    QString m_poolName;

public:
    explicit NRThreadPool(int numberOfThreads2Spawn=0,
                          const QString &poolname="NrTPool",
                          QObject *parent = 0);
    void setPolicy(ThreadAssignmentPolicy tap)
    int runObject(QObject *o, int preferred_tid=-1);
};
```

NrThreadPool (3)

```
int NRThreadPool::runObject(QObject *o, int preferred_tid)
{
    int tid;
     if (preferred_tid == -1) {
         tid = findThread2Use(); // implements the allocation policy
     }
     else {
         tid = preferred_tid;
     }
     // and now the trick :)
     o->moveToThread( m_v[tid] );
    // other stuff
     return 0;
}
```

MTServer (1)

- Template class (*but QObject and templates does not play along*)
  - NrServerWorker
- Uses **NrThreadPool** to manage workers
  - A worker ~ client/server connection
- Soft / hard connection limits
  - **Soft limit** → emit connection_exhausting()
  - **Hard limit** → emit connection_rejected()
- Socket inactivity timeout
  - readyRead() slot updates a socket to QdateTime QMap
  - Connection is closed if no data on socket for T [sec]
- Plaintext / SSL connections
- MTServer worket interface/contract
  1. **Socket from the Q[Ssl]Server is injected in the worker constuctor**
  2. **virtual void handleClientData()**

MTServer (2)

```
class QMultiThreadedServer : public QObject
{
    Q_OBJECT

    QMap<QTcpSocket*, QDateTime> m_Socket2LastTStampMap;
    QTimer *m_pTStampCheckerTimer;
    NRThreadPool *m_pTPool;

public:
    QMultiThreadedServer(const NrServerConfig &i_rSrvConf,
                         quint16 i_numberOfThreads=0,
                         QObject*parent=NULL);

signals:
    void clientConnected(NrServerWorker *);
    void clientDisconnected(NrServerWorker *);
    void clientRejected();
    void clientConnectionsExhausting(int);
};
```

MTServer (3)

```
template <class T = NrServerWorker>
class MultiThreadedServer : public QMultiThreadedServer
{
protected:
    NrServerWorker* getNewWorkerPointer(QTcpSocket*);
public:
    explicit MultiThreadedServer(const NrServerConfig &srvconf,
                                 quint16 i_numberOfThreads=0,
                                 QObject* parent=NULL);
    virtual ~MultiThreadedServer();
};

//Now include the template implementation
#include "mthreadserver.tpp"
```

MTServer (4)

```
void
QMultiThreadedServer::onNewClientConnection()
{
    QTcpSocket *sock = m_pSslServer->nextPendingConnection();
    if ( hardLimitReached() ) {
        sock->abort();
        sock->deleteLater();
        emit clientRejected();
        return;
    }
    if ( softLimitReached() ) {
        emit clientConnectionsExhausting(cc);
    }
    NrServerWorker *wo = getNewWorkerPointer(sock);
    sock→setParent(wo);
    // do some other stuff
    m_pTPool->runObject(wo);
}
```

MTServer (4)

```
void
QMultiThreadedServer::onNewClientConnection()
{
    QTcpSocket *sock = m_pSslServer->nextPendingConnection();
    if ( hardLimitReached() ) {
        sock->abort();
        sock->deleteLater();
        emit clientRejected();
        return;
    }
    if ( softLimitReached() ) {
        emit clientConnectionsExhausting(cc);
    }
    NrServerWorker *wo = getNewWorkerPointer(sock);
    sock→setParent(wo);
    // do some other stuff
    m_pTPool->runObject(wo);
}
```

- So far so good: **~600 LOC** (tpool + mtserver)

ApiRestServer (1)

- Basically a MTServer<ApiRestWorker> + signal/slot to glue things together
  - **Not a fully featured HTTP/REST Server**
- Api reply timeout
- QUuid to manage reply recollection
- Signals to communitate the REST received method
  - Supported methods: **GET**/**POST**/**DELETE**
  - the actual heavy lifting is done somewhere else

ApiRestServer (2)

```
class ApiRestServer : public QObject
{
    Q_OBJECT
    MultiThreadedServer<ApiRestWorker>* m_apiRestServerPtr;

public:
    void sendReplyToClient(int i_statusCode,
                           const QByteArray& i_replyMessage,
                           const QUuid &i_uuid);

signals:
   void getRequestSignal(const QString& i_url, const QUuid& i_uuid);
   void postRequestSignal(const QString& i_url,
                          const QByteArray& i_postMessage,
                          const QUuid& i_uuid);
   void deleteRequestSignal(const QString& i_url,
                            const QUuid& i_uuid);
```

ApiRestWorker (1)

- Receives data from the client's socket
  - Sends data to a **DataSlicer**
  - TCP is a stream-oriented protocol
- **Qurl + QRegExp** to parse received message

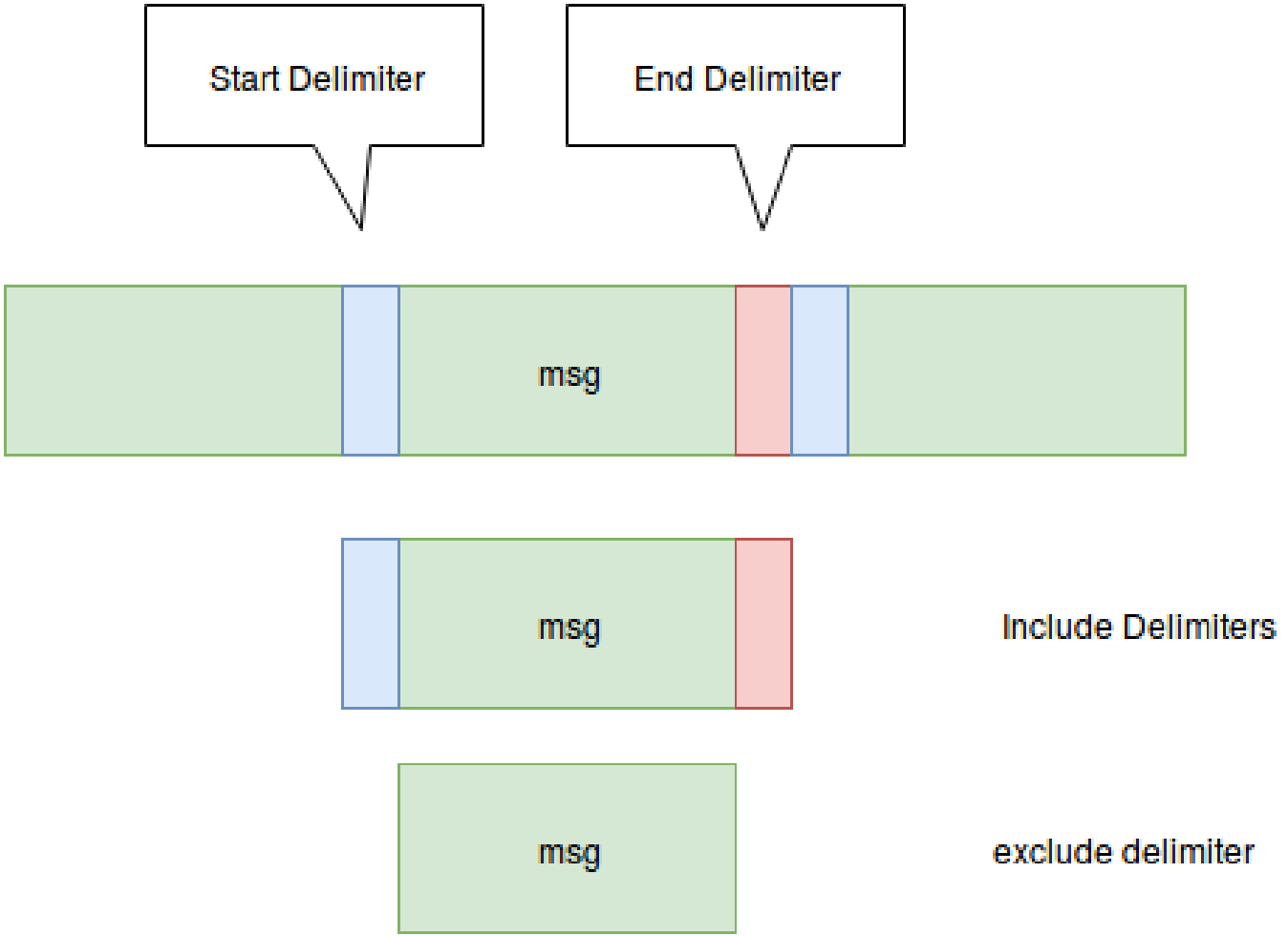
ApiRestWorker (2)

```
class ApiRestWorker : public NrServerWorker
{
    Q_OBJECT

    QSharedPointer<DataSlicer>      m_messageSlicerPtr;
    QUuid                           m_uuid;

private slots:
    virtual void handleClientData() {
       QByteArray dataReceived = m_sock->readAll();
       *m_messageSlicerPtr << dataReceived;
    }

   // …
};
```

DataSlicer(1)

- DataSlicer is a configurable stream slicer
- Define one or more
  - Start & End tokens
  - Slicing Policy
    - Delimiter included/excluded
- Throw data in using **<<** operator
- emit newMessage(QByteArray) signal

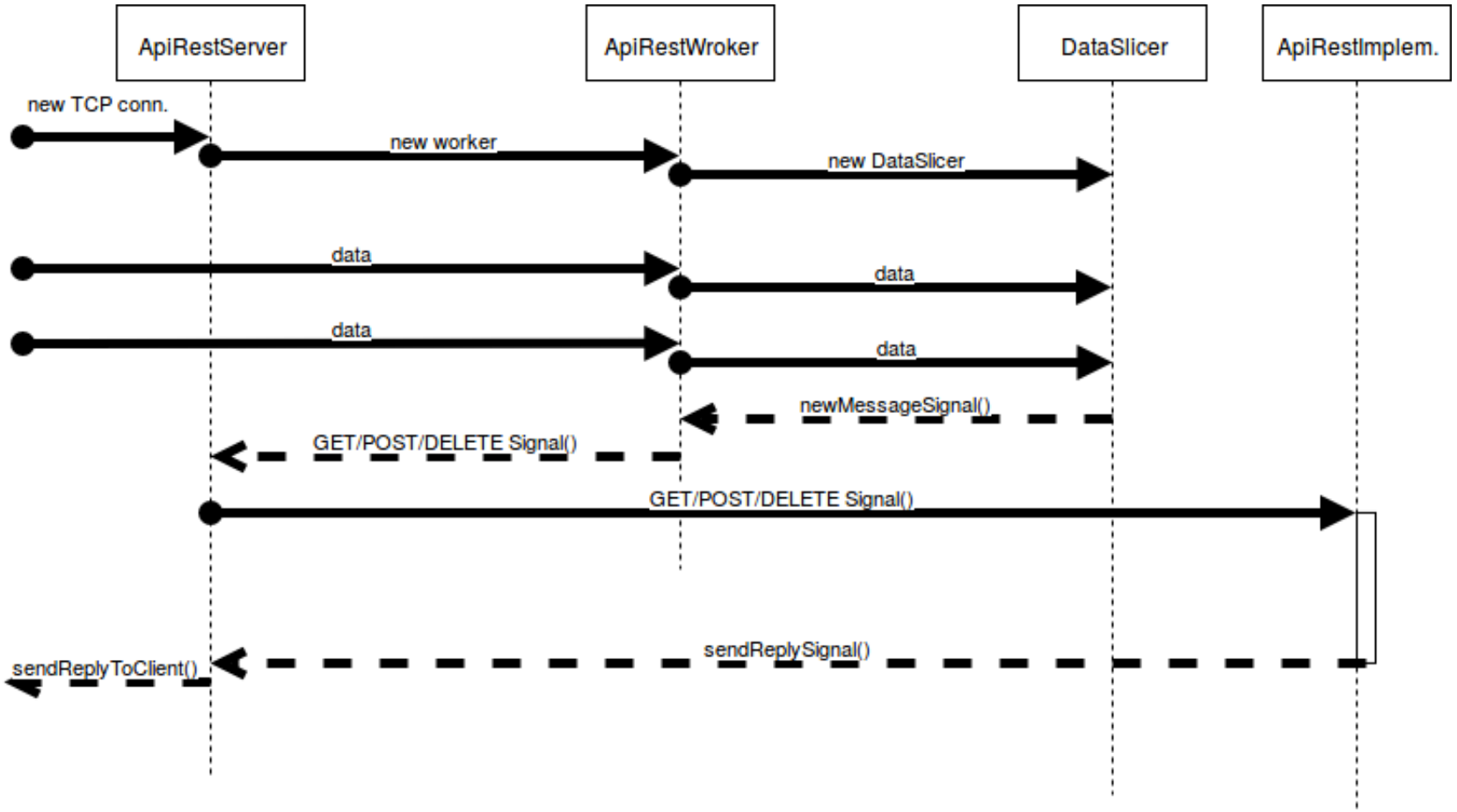
DataSlicer(2)



DataSlicer (3)

```
TokenSetStruct *tokenGET, *tokenDELETE;

tokenGET.startCondPtr = new TokenFinder(
                            apiRestMsgNs::getMethodString.toUtf8()));
tokenGET.EndCondPtr = new TokenFinder(
                            apiRestMsgNs::endOfLine.toUtf8()));
tokenGET.slicingPolicy = DELIMITER_INCLUDED;

QList<TokenSetStruct> tokenList;
    tokenList.append(tokenGET);
    tokenList.append(tokenDELETE);

m_messageSlicerPtr = QsharedPointer<DataSlicer>(
     new DataSlicer(tokenList));

connect(m_messageSlicerPtr.data(),
        SIGNAL(newMessage(QByteArray)),
        this,
        SLOT(onRequestMessageReceivedSlot(QByteArray)));
```

Putting All Together (1)



Will it fly?