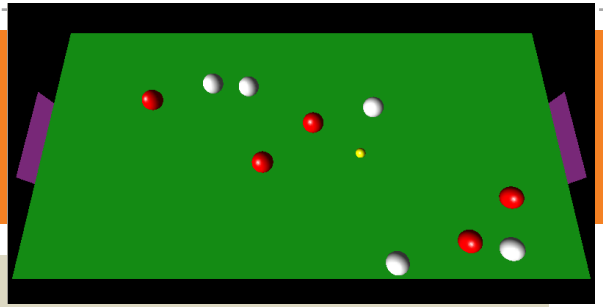
Intro to ZMQ and Google Protobuf in C++

@diegorlosada

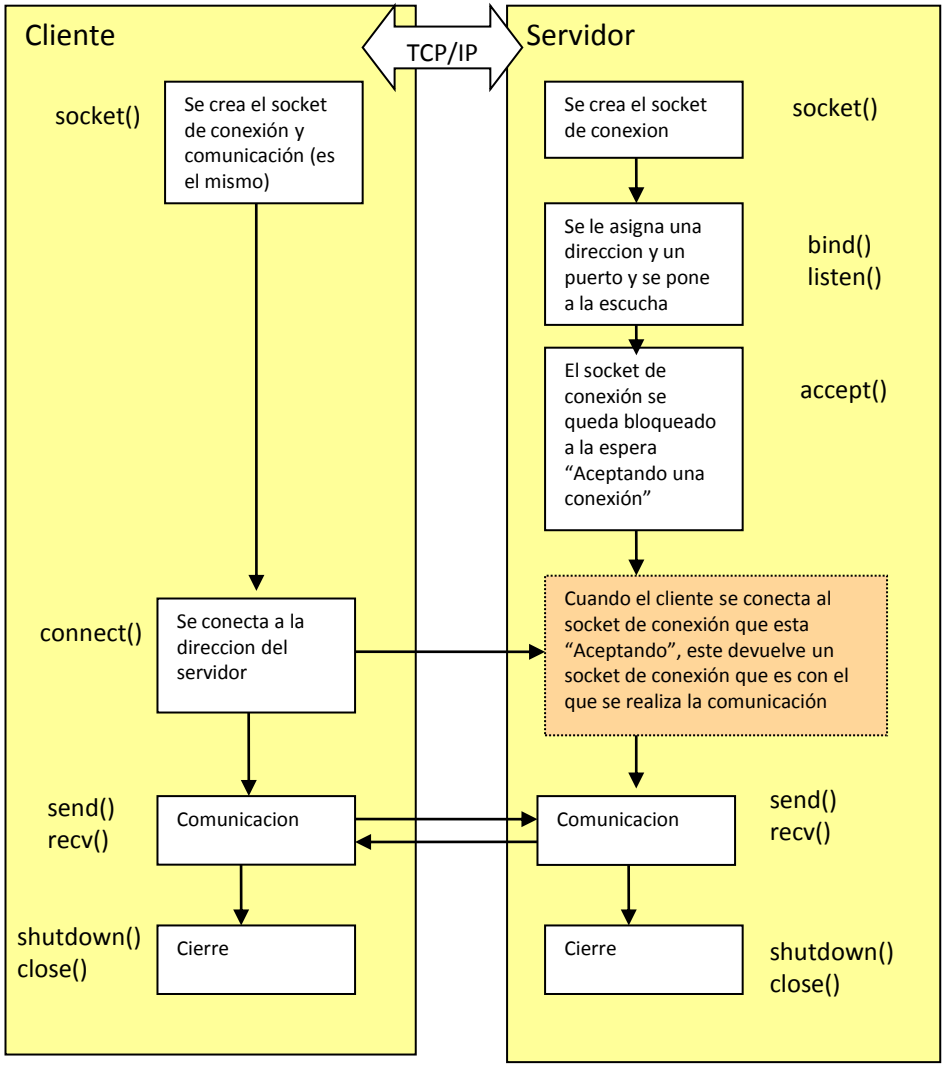
MeetupC++ March-2015

SOCCER TIME



```
$ git clone
https://github.com/drodri/soccer.git
$ cd soccer
$ bii init –L
$ bii find
$ bii configure –G “Visual Studio 12” (optional)
$ bii build
$ bin/…server
$ bin/… client (other console)
$ git co enunciado
```

Raw Sockets


Cliente
TCP/IP
Servidor
socket()
Se crea el socket de conexión y comunicación (es el mismo)
Se crea el socket de conexion
socket()
Se le asigna una direccion y un puerto y se pone a la escucha
bind()
listen()
El socket de conexión se queda bloqueado a la espera "Aceptando una conexión"
accept()
connect()
Se conecta a la direccion del servidor
Cuando el cliente se conecta al socket de conexión que esta "Aceptando", este devuelve un socket de conexión que es con el que se realiza la comunicación
send()
recv()
Comunicacion
Comunicacion
send()
recv()
shutdown()
close()
Cierre
Cierre
shutdown()
close()

ZeroMQ

High level messaging: RabbitMQ

Low level sockets

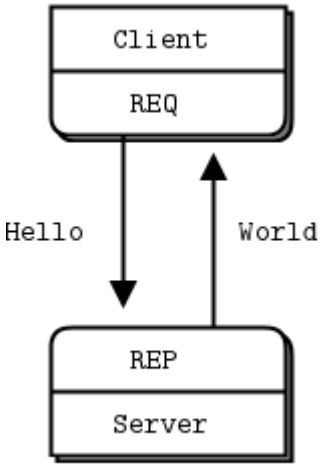
Characteristics:

- No neutral carrier
- No protocol
- License: LGPL

Other (nanomsg)

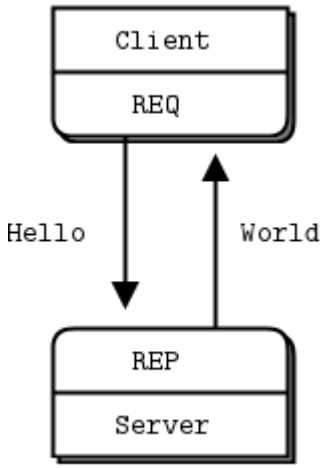
- Licence: MIT
- Pure C
- Cleaner API
- Internals efficient:
  - Zero-copy
  - Thread-safety
  - Patricia trie
- Cons:
  - Not as stable, mature

REQ-REP (Client) => RPC



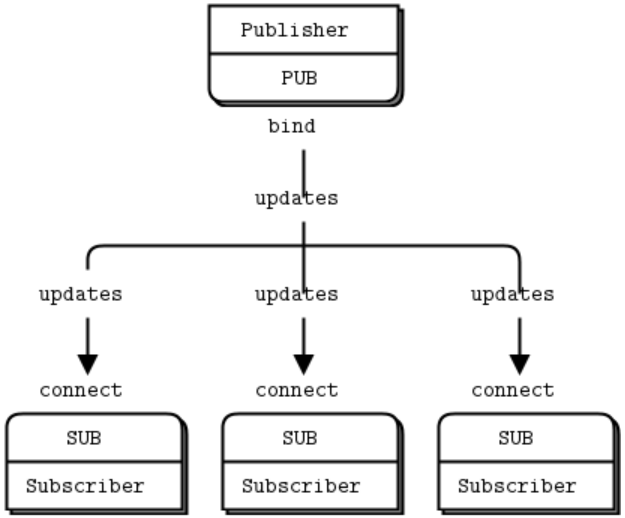

```cpp
//  Hello World client in C++
//  Connects REQ socket to tcp://localhost:5555
//
#include "diego/zmqcpp/zmq.hpp"
#include <string>
#include <iostream>

int main (){
    //  Prepare our context and socket
    zmq::context_t context (1);
    zmq::socket_t socket (context, ZMQ_REQ);
    socket.connect ("tcp://localhost:5555");

    zmq::message_t request (6);
    memcpy ((void *) request.data (), "Hello", 5);
    socket.send (request);

    //  Get the reply.
    zmq::message_t reply;
    socket.recv (&reply);
}
```

REQ-REP (Server) => RPC




```cpp
//
//  Hello World Server in C++
//  Connects REQ socket to tcp://localhost:5555
//
#include "diego/zmqcpp/zmq.hpp"
#include <string>
#include <iostream>

int main () {
    zmq::context_t context (1);
    zmq::socket_t socket (context, ZMQ_REP);
    socket.bind ("tcp://*:5555");

    while (true) {
        zmq::message_t request;
        socket.recv (&request);

        //  Do some 'work'

        zmq::message_t reply (5);
        memcpy ((void *) reply.data (), "World", 5);
        socket.send (reply);
    }
    return 0;
}
```

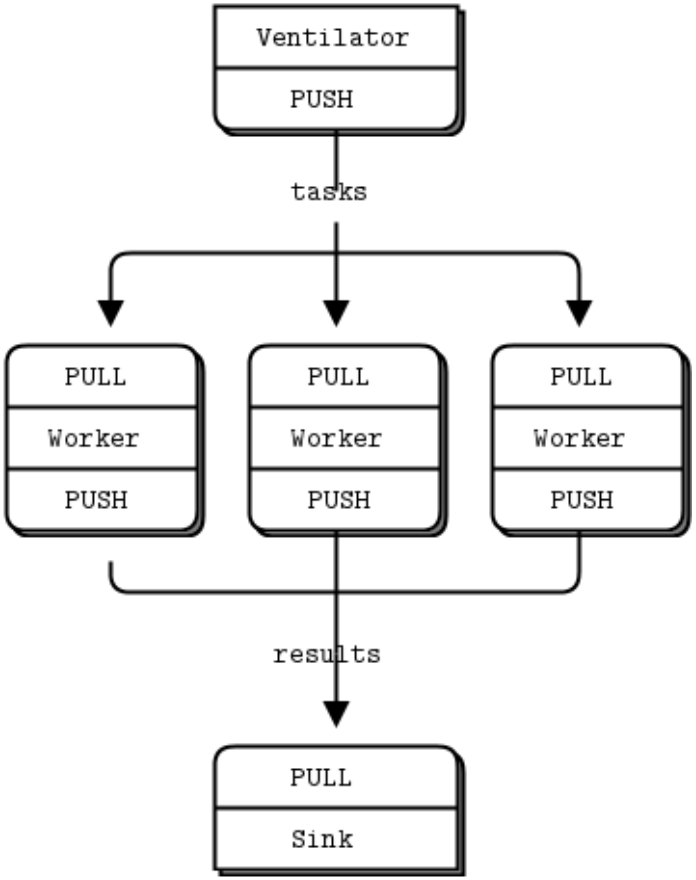
PUB-SUB => Broadcast




```
zmq::context_t context;
zmq::socket_t publisher(context, ZMQ_PUB);
publisher.bind("tcp://*:12345");

std::string status="Hello";
zmq::message_t zmsg;
get_msg(status, zmsg);
publisher.send(zmsg);
```

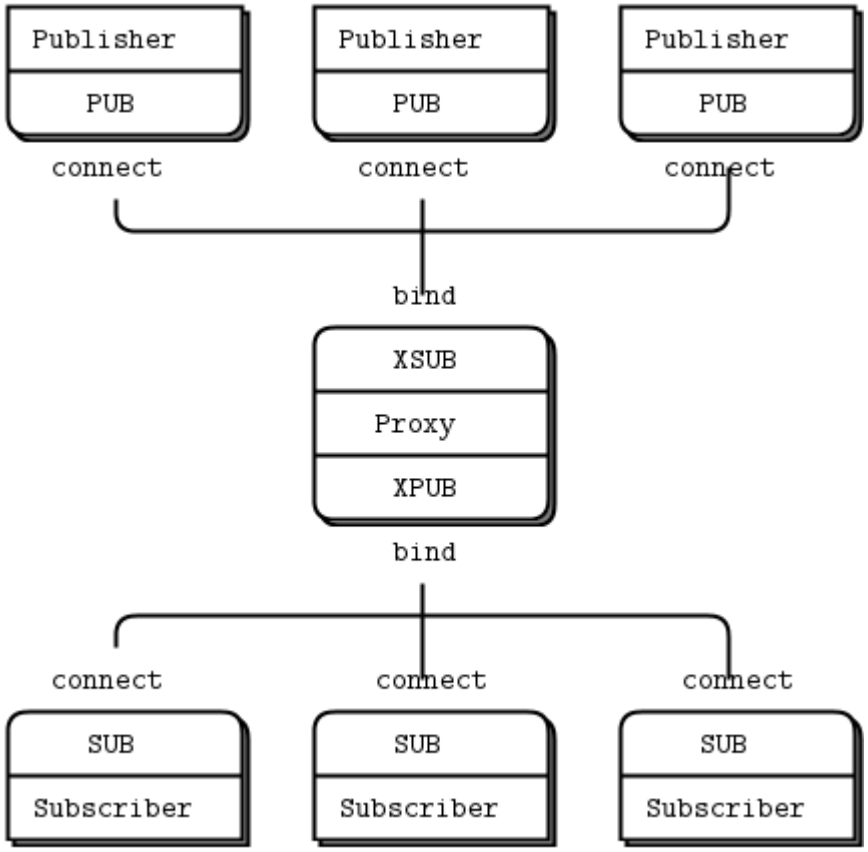
```
zmq::context_t context;
zmq::socket_t subscriber(context, ZMQ_SUB);
subscriber.connect("tcp://localhost:12345");
const char *filter = "";
subscriber.setsockopt(ZMQ_SUBSCRIBE, filter, strlen(filter));
```

PUSH-PULL => Load Balance, Fair Queueing

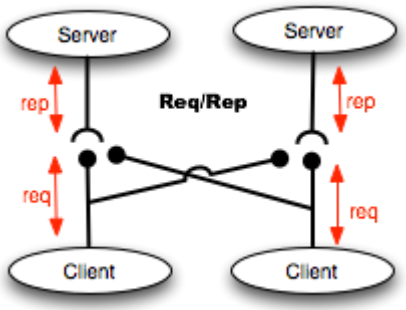


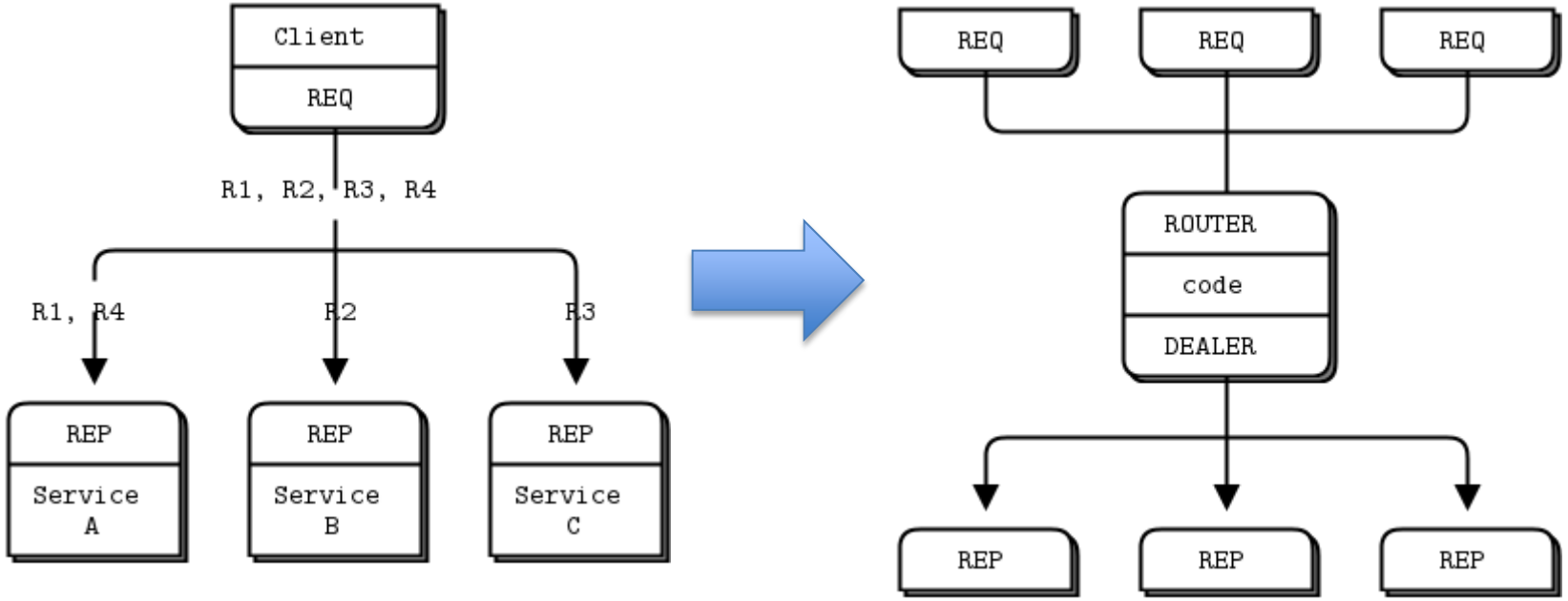

```
//CLIENT PUSH
zmq::socket_t sender(context, ZMQ_PUSH);
sender.connect("tcp://localhost:12346");


//SERVER PULL
zmq::socket_t receiver(context, ZMQ_PULL);
receiver.bind("tcp://*:12346");

zmq::message_t command;
while (receiver.recv(&command, ZMQ_NOBLOCK)){
    std::string com = get_str(command);
    std::cout<<com;
}
```

PUB-SUB with proxy



Extend REQ-REP with ROUTER-DEALER





Serialization

- JSON, XML, CSV…
  - Parsing efficiency (vs human readable)
  - Robust to changes
    - Adding fields
    - Removing fields? Ignoring fields
  - SW engineering possible:
    - Composable, hierarchies…
- Necessary for:
  - Persistence
  - Transport (network)
    - RPC/ distributed systems

Protocol Buffers (Protobuf)

- Serialization format by Google
  used by Google for almost all internal RPC protocols an d file formats
  - (currently 48,162 different message types defined in the G oogle code tree across 12,183 .proto files. They're used bot h in RPC systems and for persistent storage of data in a vari ety of storage systems.)
- XML vs Protobuf
  - Speed 20-100x
  - Size 10x
  - Not human readable
- Being there since 2001, open sourced in 2008
  - BSD

Others(Apache Thrift)

- Include RPC
- More languages
- Slower
- More containers (set, map)

Other (Cap'n'proto)

- Cap'n Proto is an insanely fast data interchange format
- RPC system. Think JSON, except binary.
- Use padding (zeroes), then compress to send over the wire, achieve < size than Protobuf

```
struct Person {
  id @0 :UInt32;
  name @1 :Text;
  email @2 :Text;
  phones @3 :List(PhoneNumber);

  struct PhoneNumber {
    number @0 :Text;
    type @1 :Type;

    enum Type {
      mobile @0;
      home @1;
      work @2;
    }
  }

  employment :union {
    unemployed @4 :Void;
    employer @5 :Text;
    school @6 :Text;
    selfEmployed @7 :Void;
    # We assume that a person is only one of these.
  }
}

struct AddressBook {
  people @0 :List(Person);
}
```

Other (Cap'n'proto)


```
struct Person {
  id @0 :UInt32;
  name @1 :Text;
  email @2 :Text;
  phones @3 :List(PhoneNumber);

  struct PhoneNumber {
    number @0 :Text;
    type @1 :Type;

    enum Type {
      mobile @0;
      home @1;
      work @2;
    }
  }

  employment :union {
    unemployed @4 :Void;
    employer @5 :Text;
    school @6 :Text;
    selfEmployed @7 :Void;
    # We assume that a person is only one of these.
  }
}

struct AddressBook {
  people @0 :List(Person);
}
```

```
#include "addressbook.capnp.h"
#include <capnp/message.h>
#include <capnp/serialize-packed.h>
#include <iostream>

void writeAddressBook(int fd) {
  ::capnp::MallocMessageBuilder message;

  AddressBook::Builder addressBook = message.initRoot<AddressBook>();
  ::capnp::List<Person>::Builder people = addressBook.initPeople(2);

  Person::Builder alice = people[0];
  alice.setId(123);
  alice.setName("Alice");
  alice.setEmail("alice@example.com");
  // Type shown for explanation purposes; normally you'd use auto.
  ::capnp::List<Person::PhoneNumber>::Builder alicePhones =
      alice.initPhones(1);
  alicePhones[0].setNumber("555-1212");
  alicePhones[0].setType(Person::PhoneNumber::Type::MOBILE);
  alice.getEmployment().setSchool("MIT");

  Person::Builder bob = people[1];
  bob.setId(456);
  bob.setName("Bob");
  bob.setEmail("bob@example.com");
  auto bobPhones = bob.initPhones(2);
  bobPhones[0].setNumber("555-4567");
  bobPhones[0].setType(Person::PhoneNumber::Type::HOME);
  bobPhones[1].setNumber("555-7654");
  bobPhones[1].setType(Person::PhoneNumber::Type::WORK);
  bob.getEmployment().setUnemployed();

  writePackedMessageToFd(fd, message);
}

void printAddressBook(int fd) {
  ::capnp::PackedFdMessageReader message(fd);
```

Protocol Buffers IDL

message.proto

```
package tutorial;

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
```

Generate code

message.proto

```
package tutorial;

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
```

message.pb.h

```
package tutorial;

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
```

message.pb.cc

```
package tutorial;

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
```

```
$ bin/protoc message.proto --cpp_out=“.”
```

Use

```cpp
#include <iostream>
#include <sstream>
#include <string>
#include "message.pb.h"

using namespace std;


int main() {
  // Verify that the version of the library that we linked against is
  // compatible with the version of the headers we compiled against.
  GOOGLE_PROTOBUF_VERIFY_VERSION;

  tutorial::AddressBook address_book;
  tutorial::Person* person = address_book.add_person();

  person->set_id(123);
  person->set_name("John");
  person->set_email("john@gmail.com");
  tutorial::Person::PhoneNumber* phone_number = person->add_phone();
  phone_number->set_number("1234567");
  phone_number->set_type(tutorial::Person::MOBILE);

  ostringstream output;
  address_book.SerializeToOstream(&output);

  istringstream input(output.str());
  tutorial::AddressBook address_book2;
  address_book2.ParseFromIstream(&input);

  cout<<address_book2.DebugString();
```

Protocol Buffers IDL

- bool SerializeToString(string* output) const;
- bool ParseFromString(const string& data);
- bool SerializeToOstream(ostream* output) const;
- bool ParseFromIstream(istream* input);

Intro to ZMQ and Google Protobuf in C++

@diegorlosada

MeetupC++ March-2015