Referee Box for the RoboCup Logistics League

Integrator Manual

Tim Niemueller
niemueller@kbsg.rwth-aachen.de

March 8, 2013

1 Introduction

Under the umbrella of RoboCup, in 2012 the first competition in the *Logistics League sponsored by Festo* (LLSF) took place. The idea of this new RoboCup league is to simulate a production process where an assembly robot has to bring goods to production places to refine them step by step to a final product. A team of robots has to explore which machine is producing/refining what semi-finished products and produce as many products as possible.

Already at this early stage it became apparent, that just observing the game and awarding points is complex. Especially when introducing more goods that can be processed, and when expecting more robots operating concurrently in the arena in the future, we need to consider options to automate the evaluation of the game.

The Referee Box (refbox) controls and monitors the game to perform this very evaluation. We have tried to make the refbox as autonomous as possible. Textual and graphical user interfaces allow for human instruction and supervision. This is in particular required in unexpected situations (e.g. humans need to stop the game if a robot catches fire) or where perceptual input is not available (a puck is moved out of the are of a machine which is still waiting for more goods). Several parts of the refbox were influenced by the Fawkes Robot Software Framework [1] – some libraries like the configuration and logging facilities are simplified versions of the Fawkes libraries – and ROS [2]. The internally used CLIPS environment is based on the Carologistics' task coordination approach [3].

This manual describes how to integrate a robot system with the refbox, in particular the communication protocols involved. For detailed information about system requirements to build and run the refbox as well as configuration and usage instructions please refer to the referee box website at `http://www.robocup-logistics.org/refbox`.

2 Infrastructure

The refbox communicates with two different classes of entities, robots and controllers. Any machine of a playing team is considered a robot. Controllers are used by the human referees to interact with the refbox, visualize its internal state, and give instructions to it.

All communication is handled using IPv4. There are two primary ways of communication. First, controllers access the refbox using a TCP-based stream communication protocol in a client-server fashion. Second, robots communicate with the refbox using UDP-based broadcast protocol in a peer-to-peer fashion.

Messages are defined and encoded using the Google protobuf library. It provides easy to use, extensible, and widely compatible data exchange structures. Messages are transmitted using a minimal protocol for framing the serialized messages.

In the following we give a brief overview of Protobuf and how the messages are defined. We then introduce the framing protocol and how it is used to transmit messages. We then describe the

basic stream and broadcast communication modes and conclude with an overview and example using the protobuf_comm library which is part of the refbox.

2.1 Protocol Buffers (protobuf)

Protocol Buffers[1] (protobuf) offers a description language to define data exchange formats which can be encoded and decoded very efficiently. The formats are extensible and provide for basic data types, nesting, structure re-use, efficient serialization and deserialization, variable-length lists, and a compiler to create code for C++, Java, or Python to access the data. In this document, we will only give a very brief introduction and we refer to the protobuf website[1] for more detailed information.

Data structures are defined as a message and are somewhat similar to C/C++ data structures. Each field has a rule out of `required`, `optional`, or `repeated`, a type, a name, and a unique tag number. The rule defines if the field must or can be present, or if it can be zero, one, or more values of the given type. The type can be either one of a number of basic types, or another message type. The tags must be unique within a message. They are used to identify the fields when deserializing.

Structures in other files can be used by importing a file. See for example BeaconSignal.proto as an example in Section 3. After defining the messages in proto files, they must be compiled to generate the actual code to use the data structures. Protobuf comes with support for C++, Java, and Python out of the box. For other languages third-party tools exist. The code must then be compiled into a library or your application.

2.2 Framing Protocol

Once protobuf messages are serialized, they contain no information about the contained type or the length of the message. Therefore, to transmit them over the network a framing protocol is required. This adds a little frame around each message indicating its type and size to allow for transmitting arbitrary messages over the same connection. This is provided by the framing protocol.

```
typedef struct {
  /** component id */
  uint16_t  component_id;
  /** message type */
  uint16_t  msg_type;
  /** payload size in bytes */
  uint32_t  payload_size;
} frame_header_t;
```

The structure shown to the right describes the frame header. Each message that is sent over the network is prepended such a header. Each header consists of three values. The component ID can be used to dispatch messages to a particular component. In our case, all messages have a component ID of 2000 for the refbox. The message type denotes the contained protobuf message and determines how the message is deserialized. The payload size specifies the amount of data to read for the next message for deserialization.

The message must be 4-byte-aligned, i.e. the message structure when sent over the network must have a size of exactly eight bytes. All contained numbers must be encoded in network byte order (big endian, most significant bit first). The numbers are encoded as 16 (`uint16_t`) or 32 bit (`uint32_t`) unsigned integers respectively.

When sending a message over the network, first the protobuf message is serialized (to determine the payload size). Then the frame header is prepared with the appropriate component ID, message type, and the payload size as just determined. Then the frame header is sent immediately followed by the serialized message.

Note that for UDP packets, the data must be serialized into a common buffer. Otherwise the operating systems network stack could split the transmission into two UDP messages, one containing the frame header and the other the message. This is invalid and cannot be decoded on the other end (UDP does not provide for receiving in the correct order and packet correlation).

[1] `https://code.google.com/p/protobuf/`

2.3 Communication Modes

The refbox uses two communication modes. A TCP-connection-based streaming protocol is used for communication between the refbox and controllers like the shell or GUI. Teams may not use this mode to contact the refbox. For such communication a UDP-based broadcast protocol is employed.

Since the wifi is inherently unreliably, in particular during a RoboCup competition. Hence connections can be lost at any time. In particular on a busy wireless network a TCP handshake can mean a high performance penalty, or can even mean reliable communication fails due to frequent re transmissions. Therefore, for communication with the robots a UDP-based protocol was chosen. For now, we use broadcasting to communicate the information to all robots at once. In the future, this might be extended to Multicast.

To implement stream-based communication, we recommend to implement it in such a way that always the frame header is read first, and only afterwards the payload is read depending on the size given in the header.

For broadcast-based communication, make sure to build a complete message buffer including the frame header and the serialized message, and then send it in one go to avoid fragmentation. When reading, create a sufficiently large buffer to read the full package at once. Then read the frame header from the buffer and deserialize the message.

The refbox accepts stream client connections on TCP port 4444. During the tournament, connections will most likely only be allowed from the local or selected hosts. Neither a robot nor any other team's device may use the stream protocol to communicate with the refbox. Peer-to-peer broadcast communication is down on UDP port 4444. The refbox will communicate on an assigned broadcast address determined at the tournament. Make sure that all of this information is easily configurable on your system.

2.4 Example using protobuf_comm

The protobuf_comm library provides an implementation of network communication with protobuf messages using the framing protocol described above for both communication modes. It is available as part of the refbox source code[2] in the `src/libs/protobuf_comm` directory. The implementation uses Boost Asio for the asynchronous I/O implementation, and Boost Signals2 to provide events that your code can connect to. The protobuf_comm library currently requires a compiler accepting code according to the C++11 standard, e.g. GCC 4.6 or higher.

The library has three principal classes of interaction that you may or may not use. First, the ProtobufStreamClient class implements a TCP socket connection using the framing protocol to receive and transmit messages from and to a ProtobufStreamServer, that is run in the refbox. Second, the ProtobufBroadcastPeer implements peer-to-peer communication over UDP using the framing protocol. Finally, the MessageRegister is employed by both classes to register messages you wish to react to. Messages types must be registered with the message register so that the client or peer can now how to deserialize incoming messages. Messages of an unknown type are ignored.

The library uses the signal-slot event pattern to notify your code about events such as being connected or disconnected, or receiving an incoming message. This is built on top of Boost.Signals2[3]. Note that this is incompatible with Boost.Signals. Signals2 was chosen because it provides an increased thread-safety. The ProtobufStreamClient and ProtobufBroadcastPeer run a concurrent thread to process incoming data. So when a signal is issued, it happens in the context of the receiving thread. For code that is not thread-safe, or that expects events only from a particular thread (e.g. GUI applications using Gtkmm or even text interfaces using ncurses), you need to implement a dispatch pattern that will issue an event that is evaluated in that main thread.

In Figure 1 you see the code for an example how to use peer communication. You can also find the code in the `src/examples` directory in the refbox source. You can adapt the code and

[2]Instructions how to obtain the code at `http://trac.fawkesrobotics.org/wiki/LLSFRefBox/Install`

[3]Cf. `http://www.boost.org/libs/signals2/`

```
#include <protobuf_comm/peer.h>
#include <msgs/GameState.pb.h>

using namespace protobuf_comm;

class ExamplePeer
{
 public:
  ExamplePeer(std::string host, unsigned short port)
  {
    peer_ = new ProtobufBroadcastPeer(host, port);

    MessageRegister & message_register = peer_->message_register();
    message_register.add_message_type<llsf_msgs::GameState>();

    peer_->signal_error().connect(
      boost::bind(&ExamplePeer::peer_error,
                  this, boost::asio::placeholders::error));
    peer_->signal_received().connect(
      boost::bind(&ExamplePeer::peer_msg, this, _1, _2, _3, _4));
  }


  ~ExamplePeer()
  {
    delete peer_;
  }

 private:
  void peer_error(const boost::system::error_code &error)
  {
    printf("Peer error: %s\n", error.message().c_str());
  }

  void peer_msg(boost::asio::ip::udp::endpoint &endpoint,
                uint16_t comp_id, uint16_t msg_type,
                std::shared_ptr<google::protobuf::Message> msg)
  {
    std::shared_ptr<llsf_msgs::GameState> g;
    if ((g = std::dynamic_pointer_cast<llsf_msgs::GameState>(msg))) {
      printf("GameState received from %s: %u points\n",
             endpoint.address().to_string().c_str(), g->points());
    }
  }

 private:
  ProtobufBroadcastPeer *peer_;
};
```

Figure 1: Example program for a protobuf_comm peer

```
#include <protobuf_comm/client.h>
#include <msgs/GameState.pb.h>

using namespace protobuf_comm;

class ExampleClient
{
 public:
  ExampleClient(std::string host, unsigned short port)
    : host_(host), port_(port)
  {
    client_ = new ProtobufStreamClient();

    MessageRegister & message_register = client_->message_register();
    message_register.add_message_type<llsf_msgs::GameState>();

    client_->signal_connected().connect(
      boost::bind(&ExampleClient::client_connected, this));
    client_->signal_disconnected().connect(
      boost::bind(&ExampleClient::client_disconnected,
                  this, boost::asio::placeholders::error));
    client_->signal_received().connect(
      boost::bind(&ExampleClient::client_msg, this, _1, _2, _3));

    client_->async_connect(host.c_str(), port);
  }


  ~ExampleClient()
  {
    delete client_;
  }

 private:
  void client_connected()
  { printf("Client connected\n"); }

  void client_disconnected(const boost::system::error_code &error)
  {
    printf("Client DISconnected\n");
    usleep(100000);
    client_->async_connect(host_.c_str(), port_);
  }

  void client_msg(uint16_t comp_id, uint16_t msg_type,
                  std::shared_ptr<google::protobuf::Message> msg)
  {
    std::shared_ptr<llsf_msgs::GameState> g;
    if ((g = std::dynamic_pointer_cast<llsf_msgs::GameState>(msg))) {
      printf("GameState received: %u points\n", g->points());
    }
  }

 private:
  ProtobufStreamClient *client_;
  std::string host_;
  unsigned short port_;
};
```

Figure 2: Example program for a protobuf_comm client

integrate it into your existing system. The full source code can be compiled to a stand-alone program for testing. However, you should integrate the code with your own main loop rather than copying the stub from the example.

In the ExamplePeer class's constructor the peer is created (line 11). In lines 13–14 the MessageRegister of the peer is retrieved, and a single message type is registered. Note that no component ID or message type are provided. They are retrieved from the CompType enum field within the message specification (cf. Section 3). In lines 16–20 the signals of the peer are connected to methods of the ExamplePeer class. The events can be issued as soon as the signal is connected. So make sure any required resource initialization has been completed or is guarded when connecting the signals. The ExamplePeer's `peer_error` (line 30ff.) and `peer_msg` (line 35ff.) methods are called if a message reception failed or succeeded respectively. Since the UDP communication is connection-less no connected or disconnected signals exist. To detect that the refbox is reachable listen to incoming BeaconSignal messages (see below). In the `peer_msg` method there are two ways to decide what message has been received. You can either decide based on the component ID and message type numbers (always both!), or you can use C++ Runtime Type Information (RTTI). The latter is preferred and shown in the example. It provides strong typing guarantees. As always, do not forget to free the peer's resources by deleting it when no longer required (destructor in line 24ff.).

In the ExampleClient class's constructor the client is created (line 12). In lines 14–15 the MessageRegister of the client is retrieved, and a single message type is registered. Note that no component ID or message type are provided. They are retrieved from the CompType enum field within the message specification (cf. Section 3). In lines 16–23 the signals of the client are connected to methods of the ExampleClient class. The events can be issued as soon as the client is connected. The `async_connect` method of the client will trigger connecting to the server, but it does not block. Make sure any required resource initialization has been completed or is guarded when connecting the client. Once the connection has been established, the `connected` signal is issued and thus the ExampleClient's `client_connected` method (line 35ff.) is called. Likewise, when the connection is lost the `client_disconnected` method (line 38ff.) is called with an error code indicating the reason of the error. In the `client_msg` method there are two ways to decide what message has been received. You can either decide based on the component ID and message type numbers (always both!), or you can use C++ Runtime Type Information (RTTI). The latter is preferred and shown in the example. It provides strong typing guarantees. As always, do not forget to free the peer's resources by deleting it when no longer required (destructor in line 29ff.).

3 Messages

In this section we describe the actual messages used for communication. They are transmitted using the infrastructure described above in accordance to the rules and regulations described in the LLSF rule book.

Messages require a component ID and message type. The tuple must be unique among all messages. The refbox uses the component ID 2000 for all its messages. Messages are grouped in one file if they are directly related. Within one file, one block of ten message type IDs is used (with the principal messages BeaconSignal, AttentionMessage, and VersionInfo being an exception).

The message types are encoded in the message proto files as a CompType enum sub-type per message. The CompType must always be of the same form, having a COMP_ID and a MSG_TYPE entry. The COMP_ID is assigned the component ID number, i.e. 2000 for the refbox. The MSG_TYPE is assigned the type identification number of the particular message. It must be unique among all LLSF refbox messages. A message type number may not be re-used once a message type is removed, as not to confuse older peers expecting the old message type.

In the following, we give the protobuf specifications and description of all message types currently used for the LLSF refbox communication. The "Via" field describes what communication method is used to transmit the message, it can be either Peer-to-Peer (P2P), Client-Server (C-S), or both. Many messages are sent at certain periods. These are given in seconds time in between

two sent messages, or event if it is sent on particular events. Some broadcast messages feature a burst mode to send a number of packets of the same content in short succession to increase the chance of reaching the other peers. The "Sent by" and "Sent to" fields indicate the sender and receiver of a message. A controller is the refbox GUI or shell to communicate to, not a team's device. All proto files are in the `src/msgs` directory of the refbox source code. Feel free to copy them to your own source code, but in this case make sure to watch for updates to the messages and integrate those changes[4].

```
package llsf_msgs;

message AttentionMessage {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 2;
  }

  // The message to display, be brief!
  required string message = 1;
  // Time in sec the message is visible
  optional float  time_to_show = 2;
}
```

File: `AttentionMessage.proto`

Via:	Client-Server	**Period:**	events
Sent by:	refbox	**Sent to:**	controller

An AttentionMessage is sent when a particular event requires human attention, e.g. if a late order puck must be placed or if communication to a robot is lost.

```
package llsf_msgs;

import "Time.proto";
import "Pose2D.proto"

message BeaconSignal {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 1;
  }

  // Local time in UTC
  required Time   time = 1;
  // Sequence number
  required uint64 seq = 2;

  // The robot's team name
  required string team_name = 4;
  // The robot's name
  required string peer_name = 5;

  // Position and orientation of the
  // robot on the LLSF playing field
  optional Pose2D pose = 7;
}
```

File: `BeaconSignal.proto`

Via:	Peer-to-Peer	**Period:**	1 sec
Sent by:	all	**Sent to:**	any

The BeaconSignal is periodically sent to detect robots and the refbox on the network. If a robot is initially detected the time value is used to compare the clock offset (*not* taking network delays into account). The controller may issue a warning if this is too large. The team and robot name are required to be able to diagnose possible connection problems.

If a robot is not seen for more than 5 seconds it is considered lost, and after 30 seconds definitely lost.

We encourage teams to provide the known position of the robot in the pose field to use it for the game visualization. Not only can it make the job of the referee easier, but it will make the games more interesting to watch for visitors.

```
package llsf_msgs;

message VersionInfo {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 3;
  }

  required uint32 version_major  = 1;
  required uint32 version_minor  = 2;
  required uint32 version_micro  = 3;
  required string version_string = 4;
}
```

File: `VersionInfo.proto`

Via:	P2P & C-S	**Period:**	event
Sent by:	refbox	**Sent to:**	any

The VersionInfo message is sent to each newly connected stream client as the first message. Additionally, whenever a new peer is detected on the network, the version info is periodically sent 10 times at a period of 0.5 sec. The information can be used to ensure compatibility and warn after future refbox upgrades.

[4] Register at `https://lists.kbsg.rwth-aachen.de/listinfo/llsf-refbox-commits` to the llsf-refbox-commits mailing list to be notified of any pushes to the refbox repository.

```
package llsf_msgs;

// Time stamp and duration structure.
// Can be used for absolute times or
// durations alike.
message Time {

  // Time in seconds since the Unix epoch
  // in UTC or seconds part of duration
  required int64  sec = 1;

  // Nano seconds after seconds for a time
  // or nanoseconds part for duration
  required int64 nsec = 2;
}
```

File: `Time.proto`

Via: — **Period:** —
Sent by: — **Sent to:** —

This structure is used as a sub-message type within other messages and is never send by itself. It describes a point in time or a duration. What is sent depends on the contextual message.

A point in time is given relative to the Unix epoch in UTC time, for example using the `clock_gettime()` POSIX API call, or using Boost's `posix_time::universal_time()` method.

A duration is simply given as the number of seconds since the start and the number of nanoseconds since the start of the second.

```
package llsf_msgs;

import "Time.proto";

// Pose information on 2D map
// Data is relative to the LLSF field frame
message Pose2D {
  // Time when this pose was measured
  required Time   timestamp = 1;

  // X/Y coordinates in meters
  required float  x = 2;
  required float  y = 3;
  // Orientation in rad
  required float  ori = 4;
}
```

File: `Pose2D.proto`

Via: — **Period:** —
Sent by: — **Sent to:** —

This structure is used as a sub-message type within other messages and is never send by itself. It describes a position and orientation on the 2D ground plane of the LLSF arena. The coordinate reference frame is aligned as described in the rule book.

```
package llsf_msgs;

import "MachineInfo.proto";

message ExplorationSignal {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 70;
  }

  // machine type of this assignment
  required string    type   = 1;
  // Light specification (MachineInfo.proto)
  repeated LightSpec lights = 2;
}

message ExplorationInfo {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 71;
  }

  // The signal assignments
  repeated ExplorationSignal signals = 1;
}
```

File: `ExplorationInfo.proto`

Via: Peer-to-Peer **Period:** 1 sec
Sent by: refbox **Sent to:** robots

This ExplorationInfo message is periodically sent during the exploration phase. It announces the mapping from type names to light signals to the robots. There will be one entry per machine type in the signals list. The lights list in the ExplorationSignal sub-message will contain one entry per light color, even for signals which are off.

If one or more robots are presumably not receiving this message it is recommended to set the game state to PAUSED for some time. The ExplorationInfo message is continued to be sent in this state to allow the robot to catch up without jeopardizing the game.

```
package llsf_msgs;

import "Time.proto";

message GameState {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 20;
  }

  enum State {
    INIT = 0;
    WAIT_START = 1;
    RUNNING = 2;
    PAUSED = 3;
  }

  enum Phase {
    PRE_GAME = 0;
    EXPLORATION = 1;
    PRODUCTION = 2;
    POST_GAME = 3;
  }

  // Time in seconds since game start
  required Time   game_time = 1;

  // Current game state
  required State  state = 3;
  // Current game phase
  required Phase  phase = 4;
  // Awarded points
  optional uint32 points = 5;
}

// Request setting of a new game state
message SetGameState {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 21;
  }
  // The new desired state
  required GameState.State state = 1;
}

// Request setting of a new game phase
message SetGamePhase {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 22;
  }
  // The new desired phase
  required GameState.Phase phase = 1;
}
```

File: `GameState.proto`

Via:	P2P & C-S	**Period:**	1 sec
Sent by:	refbox	**Sent to:**	all

The GameState message is periodically sent via peer-to-peer to all robots as well as by client-server communication to connected controllers. The game time is restarted in the exploration and production phases, i.e. the time goes from 0 to 180 seconds in the exploration phase and from 0 to 900 seconds in the production phase.

In the PRE_GAME and POST_GAME phases as well as in states other than RUNNING the robots must stand still immediately and all the time.

The SetGameState and SetGamePhase messages can be used to request setting a new game state or phase respectively. They may only be sent by a controller. Be careful, setting a new phase will trigger re-initialization of that phase. To interrupt a game set the state to PAUSE, only modify the phase if you want to change irreversibly. It is illegal to send these messages from a robot. It will be detected by the refbox and considered a fraud attempt.

```
package llsf_msgs;

import "Pose2D.proto";
import "PuckInfo.proto";

enum LightColor {
  RED = 0;
  YELLOW = 1;
  GREEN = 2;
}

enum LightState {
  OFF = 0;
  ON = 1;
  BLINK = 2;
}

message LightSpec {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 10;
  }

  // Color and state of described light
  required LightColor color = 1;
  required LightState state = 2;
}

message Machine {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 12;
  }
  // Machine name and type
  required string    name     = 1;
  optional string    type     = 2;

  // Required inputs and produced output
  // Only set for field machines T*
  repeated PuckState inputs  = 3;
  optional PuckState output  = 4;
  // Puck currently in the machine area
  repeated Puck      loaded_with = 5;
  // Puck currently under the RFID sensor
  optional Puck      puck_under_rfid = 6;

  // Current state of the lights
  repeated LightSpec lights = 7;
  // Optional pose if known to refbox
  optional Pose2D    pose = 8;
  // Only set during exploration phase
  // True if correctly reported, false
       otherwise
  optional bool      correctly_reported = 9;
}

message MachineInfo {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 13;
  }

  // List of machines states
  repeated Machine machines = 1;
}
```

File: `MachineInfo.proto`

Via:	Client-Server	**Period:**	0.25 sec
Sent by:	refbox	**Sent to:**	controllers

The MachineInfo message is sent periodically to controllers by the refbox to inform them about the current state of all machines, including their static (inputs, output, name, type, pose) and dynamic information (pucks loaded into the machine area or placed under the RFID sensor, light signals, or whether it was correctly reported during the exploration phase. The refbox may report only light signals that are on or blinking. Light color that are not mentioned should be considered to be off. The pose information may or may not be sent. It can be used for visualy aligning machine representations with the actual position.

During the game the human referee at the refbox should take great care to notice any additionally connected controller (announced in the log window). No team computer or robot is allowed to connect to the refbox as particularly the machine state information allows for cheating.

```
package llsf_msgs;

import "MachineInfo.proto";
import "PuckInfo.proto";

message MachineReportEntry {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 60;
  }

  // Machine name and recognized type
  required string name = 1;
  required string type = 2;
}

// Robots send this to announce recognized
// machines to the refbox.
message MachineReport {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 61;
  }

  // All machines already already recognized
  // or a subset thereof
  repeated MachineReportEntry machines = 1;
}


// The refbox periodically sends this to
// acknowledge reported machines
message MachineReportInfo {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 62;
  }

  // Names of machines for which the refbox
  // received a report from a robot (which
  // may have been correct or wrong)
  repeated string reported_machines = 1;
}
```

File: `MachineReport.proto`

Via:	Peer-to-Peer	**Period:**	2 sec
Sent by:	robots	**Sent to:**	refbox

During the exploration phase robots announce recognized machines with the MachineReport message stating name and types. In each message, one or more machines can be reported. We suggest do always send all recognized machines to make sure that even in the case of packet losses, all machines are reported to the refbox eventually. The message should be send periodically, for example once every two seconds.

Note that for each machine only the first report received is accepted and points are awarded according to the rule book and the correctness. It is not possible to correct a once given wrong report. Every subsequent repetition will be silently ignored.

In the case of multiple robots there is *no* need to synchronize their report to send one merged report. Instead each robot can report independently directly to the refbox. However, remember that only the first report for each machine is considered.

```
package llsf_msgs;

message Order {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 40;
  }

  // Ordered product type
  enum ProductType {
    P1 = 1;
    P2 = 2;
    P3 = 3;
  }
  // Gate to deliver to, or any
  enum DeliveryGate {
    ANY = 1;
    D1  = 2;
    D2  = 3;
    D3  = 4;
  }

  // ID and requested product of this order
  required uint32       id = 1;
  required ProductType  product = 2;

  // Quantity requested and delivered
  required uint32 quantity_requested = 3;
  required uint32 quantity_delivered = 4;

  // Start and end time of the delivery
  // period in seconds of game time
  required uint32 delivery_period_begin = 6;
  required uint32 delivery_period_end   = 7;

  // The gate to deliver to, defaults to any
  // (non-defunct, i.e. non-red light) gate
  optional DeliveryGate delivery_gate = 8
           [default = ANY];
}

message OrderInfo {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 41;
  }

  // The current orders
  repeated Order orders = 1;
}
```

File: `OrderInfo.proto`

Via:	P2P & C-S	**Period:**	5 sec
Sent by:	refbox	**Sent to:**	any

During the production phase the refbox periodically sends the current orders to all robots and controllers. Note that the list of orders in the OrderInfo message can and will change. If late orders get activated, new orders appear in the message. Also note the delivery period of orders. In a message, there will most likely be orders with an active delivery period, while others have already past or are yet to come.

If a new order is posted, the refbox will go into a short time burst mode for a few seconds. During this time, it will send 10 messages at a period of 0.5 sec. Afterwards it continues with the normal period of 5 seconds.

```
package llsf_msgs;

import "Pose2D.proto";

// Current state of a puck
enum PuckState {
  S0 = 0;
  S1 = 1;
  S2 = 2;
  P1 = 4;
  P2 = 5;
  P3 = 6;
  CONSUMED = 7;
}

message Puck {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 50;
  }

  // Puck unique ID
  required uint32    id = 1;
  // Current state of the puck
  required PuckState state = 2;

  // Optional pose information
  optional Pose2D    pose = 3;
}

message PuckInfo {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 51;
  }

  // List of all known pucks
  repeated Puck pucks = 1;
}
```

File: `PuckInfo.proto`

Via:	Client-Server	**Period:**	1 sec
Sent by:	refbox	**Sent to:**	controllers

The PuckInfo message is sent periodically by the refbox to connected controllers. It contains state information about all pucks know to the refbox. The shell uses this information for the "Pucks" section. Each puck has a unique ID and a current state.

The pose information is optional and not currently provided. It may be used in the future with a camera-based tracking system.

```
package llsf_msgs;

import "Time.proto";
import "Pose2D.proto";

message Robot {
  enum CompType {
    COMP_ID  = 2002;
    MSG_TYPE = 31;
  }

  // Name and team of the robot as it was
  // announced by the robot in the
  // BeaconSignal message.
  required string name = 1;
  required string team = 2;
  // Host from where the BeaconSignal
  // was received
  required string host = 3;
  // Timestamp in UTC when a BeaconSignal
  // was received last from this robot
  required Time   last_seen = 4;

  // Optional pose information
  optional Pose2D pose = 6;
}

message RobotInfo {
  enum CompType {
    COMP_ID  = 2002;
    MSG_TYPE = 30;
  }

  // List of all known robots
  repeated Robot robots = 1;
}
```

File: `RobotInfo.proto`

Via:	Client-Server	**Period:**	1 sec
Sent by:	refbox	**Sent to:**	controllers

The RobotInfo message is sent periodically by the refbox to connected controllers. It contains information about all robots known to the refbox. It reflects its belief gathered from processing BeaconSignal messages from the robots. It also contains a host name from where the BeaconSignal was received, for example to diagnose problems if a robot is lost.

The pose information is optional. For teams which choose to provide this information in the BeaconSignal it is forwarded to the controller for visualization.

4 Further Development

The development of the refbox is an ongoing effort. While the basis should be reasonably stable, there will be bugs that need to be fixed and the refbox will have to be adapted to future rule changes. We welcome comments and contributions when presented in a friendly manner. There are some resources you can use to help us develop the software and join the effort.

RoboCup Logistics mailing list
Please post questions and discuss issues and possible improvements on the RoboCup Logistics mailing list.
`https://lists.kbsg.rwth-aachen.de/listinfo/robocup-logistics`

Bug Reports
Bug reports should be submitted to the Fawkes Trac system for the "LLSF RefBox" component.
`http://trac.fawkesrobotics.org/wiki/LLSFRefBox/ReportAProblem`

Source Code
The source code is maintained as a git repository. See the installation instructions on how to get a copy. There is a web interface available to view the code and changes to it.
`http://trac.fawkesrobotics.org/wiki/LLSFRefBox/Install`
`http://git.fawkesrobotics.org/llsf-refbox.git`

Licenses
Parts of the LLSF refbox are licensed under the 3-clause BSD license. Some parts, especially the ones originating from Fawkes, are released under the GNU General Public License Version 2 or later with special exceptions. See the header in the source code files for details.

References

[1] Tim Niemueller, Alexander Ferrein, Daniel Beck, and Gerhard Lakemeyer. Design Principles of the Component-Based Robot Software Framework Fawkes. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, 2010.

[2] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[3] Tim Niemueller, Gerhard Lakemeyer, and Alexander Ferrein. Incremental Task-level Reasoning in a Competitive Factory Automation Scenario. In *AAAI Spring Symposium 2013 – Designing Intelligent Robots: Reintegrating AI II*, 2013.