Referee Box for the RoboCup Logistics League

Integration Manual 2014

Tim Niemueller
niemueller@kbsg.rwth-aachen.de

June 29, 2015

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. For the future, when introducing more goods that can be processed, or when expecting more robots operating concurrently, we need to consider options to automate the evaluation of the game. This is the task of the referee box [1, 2].

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 accidentally moved out of the area of a machine which is still waiting for more goods). Several parts of the refbox were influenced by the Fawkes Robot Software Framework [3] – some libraries like the configuration and logging facilities are simplified versions of the Fawkes libraries – and ROS [4]. The internally used CLIPS environment is based on the Carologistics' [5] task coordination approach [6].

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. Afterwards we

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 messages containing fields, 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 these files. 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. To use configurable encryption modes we need to indicate the used cipher. Therefore, to transmit them over the network a framing protocol is required. Our protocol consists of two parts, a protocol frame and a message header depicted on the right as a C struct. We will describe the headers and how messages are formed in more detail below, here is short overview. The first a structure carries protocol information like version, encryption cipher, and payload size. The second is the message header that indicates the protobuf message component and type type to allow for transmitting arbitrary messages over the same connection. Each message that is sent over the network is prepended by such headers. After reading the frame header, the receiver must read as many bytes as indicated by `payload_size` plus an additional IV header size if encryption is used and the cipher requires an IV (see below).

```
typedef struct {
  /// Frame header version
  uint8_t  header_version;
  /// One of PB_ENCRYPTION_*
  uint8_t  cipher;
  /// reserved for future use
  uint8_t  reserved_2;
  /// reserved for future use
  uint8_t  reserved_3;
  /// payload size in bytes
  /// includes message and
  /// header, _not_ IV
  uint32_t  payload_size;
} frame_header_t;
```

```
typedef struct {
  /// component id
  uint16_t  component_id;
  /// message type
  uint16_t  msg_type;
} message_header_t;
```

The message must be 4-byte-aligned, i.e. the size of the frame header message structure sums up to exactly eight bytes when sent over the network, the message header will be exactly four bytes. All contained numbers must be encoded in network byte order (big endian, most significant bit first). The numbers are encoded as 8 (`uint8_t`), 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 and message headers are prepared with the appropriate component ID, message type, and the payload size as just determined. Then the frame header is sent, possibly followed by an encryption IV, again followed by the message header and the the serialized message.

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

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

We will now describe the frame and message headers in detail and then give examples for encrypted and unencrypted messages.

2.2.1 Frame Header

protocol version The version of the protocol used. Currently this must be set to 2.

cipher The cipher suite used. The acceptable values are listed in the following table.

Value	Encryption mode
0x00	No encryption
0x01	AES 128 ECB
0x02	AES 128 CBC (requires 16 byte IV)
0x03	AES 256 ECB
0x04	AES 256 CBC (requires 16 byte IV)

reserved These bits are currently unused and must be set to zero.

payload size Size in bytes of the following payload. This does include the message header and the serialized protobuf message. It does *not* include an encryption IV header (if required by cipher). The payload size *must* be encoded in network byte order (big-endian).

2.2.2 Message Header

component ID General ID, general addressee of message. For refbox message must be set to 2000 (as encoded in the protobuf messages' COMP_ID field of the CompType enum. If you use the refbox framing protocol for your own messages, choose a component ID different from the refbox ID. The component ID *must* be encoded in network byte order (big-endian).

message type Numeric message ID of the specific message serialized in the payload. Must be the ID encoded in the MSG_TYPE field of the CompType enum. The message type is specific to the component ID. Different component IDs can have message of the same message type which are unrelated. The message type *must* be encoded in network byte order (big-endian).

2.2.3 Unencrypted Message

In Figure 1 you see the byte layout for an unencrypted data packet. Each row represents 4 byte of data. The protocol buffer message is simply the buffer that is created by serializing the message at hand. The component ID and message type must be set as specified by the CompType enum in the message definition, e.g. 2000 and 1 for a beacon signal message.

2.2.4 Encrypted Message

In Figure 2 you see the byte layout for an encrypted data packet. It is similar to the unencrypted packet with two key differences. First, the initialization vector is placed between the frame and message headers. And second, the message header and protobuf payload are encrypted.

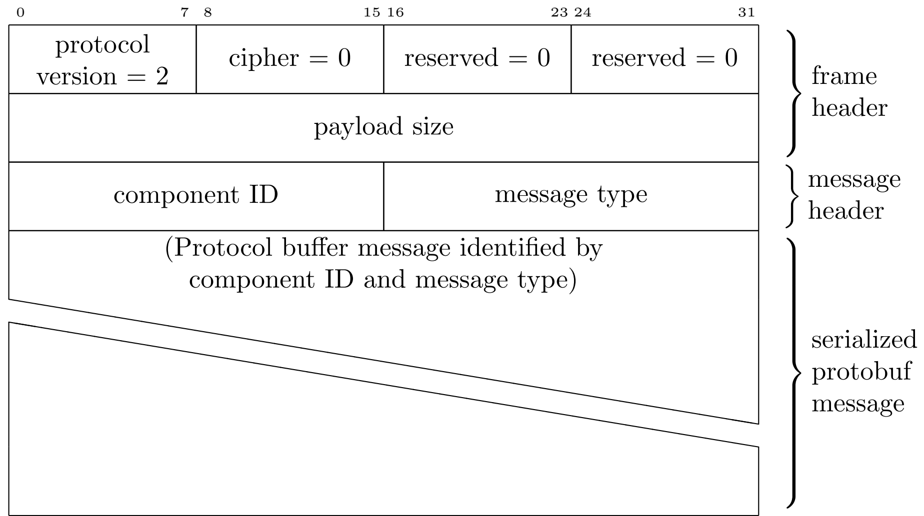

Figure 1: Packet layout diagram for unencrypted message

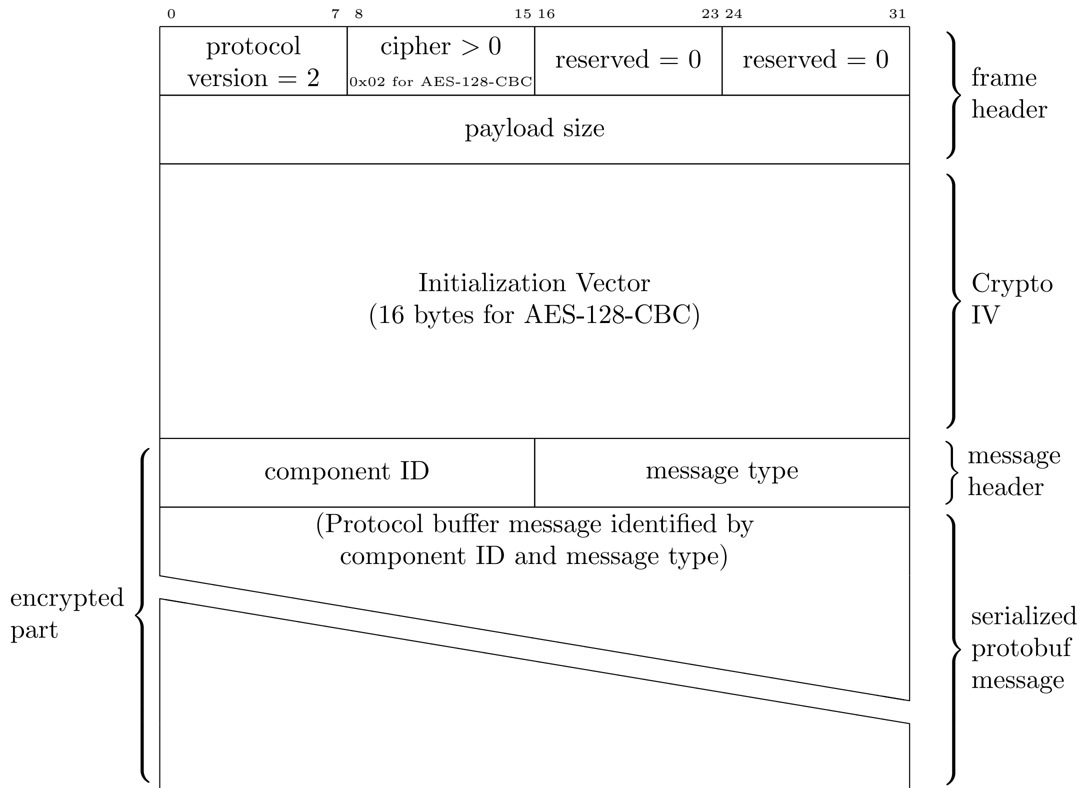

Figure 2: Packet layout diagram for encrypted message

2.2.5 Encryption Modes

The encryption currently implemented is intended to serve as a basic protection layer in particular to prevent accidental transmission of messages of the wrong team. For example, we do not insist on a time stamp within the encrypted part of the messages which would aid against replay attacks. This is done to avoid the need for time synchronization between the robots and the refbox. This might be added at a later point in time.

The framing protocol supports per-message encryption based on a symmetric block cipher. For now, the supported encryption modes are based on the AES[2] with either 128 or 256 bit key length in either electronic code book (ECB) or cipher block chaining (CBC) mode. For the tournament, the referee box will be configured to use the AES-128-CBC mode. This means that there will be a 16 byte initialization vector (IV) which is used to generate different encrypted messages for the same input (as long as different IVs are provided).

The protobuf_comm library handles this transparently for you. All you need to do is to setup encryption on the team channel broadcast peer (see below). Should you choose to create your own implementation, please still take protobuf_comm's BufferEncryptor and BufferDecryptor classes for reference. The IV must be different for each transmitted message.

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, 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.

Broadcast-based communication is further split up into three communication groups: public, team cyan, and team magenta. Generally, teams should only send on their private team channel and listen to both, the team and the public channel. The refbox will send its beacon signal and the game state on the public channel, as it is the same for both teams. Other messages like order information or exploration info is sent per team. The exact assignment is specified in the message descriptions.

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.

2.3.1 Communication Groups

The refbox differentiates a total of four communication groups. These are visualized in Figure 3. Each team needs to open two and only two broadcast peer communication channels during the game.

stream 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.

[2] Advanced Encryption Standard, cf. `http://en.wikipedia.org/wiki/Advanced_Encryption_Standard`

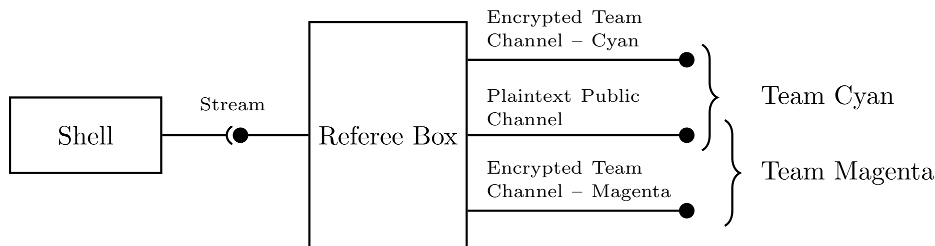

Figure 3: Communication channels and groups of the referee box

public The public broadcast channel is established on UDP port 4444. Teams may only listen to this channel, but never send.

team channels There are two team channels on UDP ports 4441 (cyan) and 4442 (magenta). Each channel will be encrypted with a team-specific encryption key that is created and provided to the teams at the beginning of the tournament (and might be re-configured later should the need arise). It is used for the private communication between the teams and the referee box.

2.3.2 Encryption Setup

Encryption on the team channels should be configured only if the refbox announced the respective team name for on of the two colors. Teams may never send messages on a private team channel if it is configured for a team different than their own.

The recommended flow for setting up encryption is like this:

- Listen to public channel for GameState messages
- If the own team name is received as the cyan or magenta team name in the GameState message, setup encryption on the respective channel
- Send messages on the team channel, never send messages on a channel of another team

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[3] in the `src/libs/protobuf_comm` directory. The implementation uses Boost Asio[4] for the asynchronous I/O implementation, and Boost Signals2[5] 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 know 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. Note that this is incompatible with Boost.Signals. Signals2 was chosen because it provides an increased thread-safety. The ProtobufStreamClient and ProtobufBroadcastPeer run

[3] Instructions how to obtain the code at `http://trac.fawkesrobotics.org/wiki/LLSFRefBox/Install`
[4] `http://www.boost.org/libs/asio`
[5] `http://www.boost.org/libs/signals2/`

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 4 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 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 13–14) which also creates a new MessageRegister that will be used. . 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++ Run-time 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++ Run-time 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

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

using namespace protobuf_comm;

#define TEAM_NAME      "Carologistics"
#define CRYPTO_KEY     "randomkey"
#define CRYPTO_CIPHER  "aes-128-cbc"

class ExamplePeer
{
 public:
  ExamplePeer(std::string host, unsigned short port)
    : host_(host), mr_(new MessageRegister()), peer_team_(NULL)
  {
    mr_->add_message_type<llsf_msgs::GameState>();
    peer_public_ = setup_peer(port, /* encrypted? */ false);
  }

  ~ExamplePeer() {
    delete peer_team_;
    delete peer_public_;
  }

 private:
  void peer_recv_error(boost::asio::ip::udp::endpoint &endpoint, std::string msg) {
    printf("Receive error from %s:%u: %s\n", endpoint.address().to_string().c_str(),
           endpoint.port(), msg.c_str());
  }

  void peer_send_error(std::string msg) {
    printf("Send error: %s\n", msg.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/%u points\n",
             endpoint.address().to_string().c_str(),
             g->points_cyan(), g->points_magenta());

      if (! peer_team_) {
        if (g->team_cyan()    == TEAM_NAME)  peer_team_ = setup_peer(4441, true);
        if (g->team_magenta() == TEAM_NAME)  peer_team_ = setup_peer(4442, true);
      } else {
        if (g->team_cyan() != TEAM_NAME && g->team_magenta() != TEAM_NAME) {
          delete peer_team_; peer_team_ = NULL;
        }
      }
    }
  }

  ProtobufBroadcastPeer * setup_peer(unsigned short int port, bool crypto) {
    ProtobufBroadcastPeer *peer = crypto
      ? new ProtobufBroadcastPeer(host_, port, mr_, CRYPTO_KEY, CRYPTO_CIPHER)
      : new ProtobufBroadcastPeer(host_, port, mr_);
    peer->signal_recv_error().connect(
      boost::bind(&ExamplePeer::peer_recv_error, this, _1, _2));
    peer->signal_send_error().connect(
      boost::bind(&ExamplePeer::peer_send_error, this, _1));
    peer->signal_received().connect(
      boost::bind(&ExamplePeer::peer_msg, this, _1, _2, _3, _4));
    return peer;
  }

 private:
  std::string           host_;
  MessageRegister       *mr_;
  ProtobufBroadcastPeer *peer_public_;
  ProtobufBroadcastPeer *peer_team_;
};

int main(int argc, char **argv) {
  ExamplePeer peer("137.226.233.255", 4444);
  while (true) usleep(100000);
}
```

Figure 4: 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/%u points\n",
             g->points_cyan(), g->points_magenta());
    }
  }

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

Figure 5: Example program for a protobuf_comm client

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 channel is used to transmit the message, it can be either public broadcast (P), team-private broadcast (T), or stream (S), or a combination thereof. 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[6]. The intention is to keep changes at a minimum, especially when approaching a tournament. But in certain situations may be necessary, e.g. to fix problems.

```
package llsf_msgs;

import "Team.proto";

option java_package =
    "org.robocup_logistics.llsf_msgs";
option java_outer_classname =
    "AttentionMessageProtos";

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

  // The message to display, be brief!
  required string message = 1;
  // Time (sec) the msg should be visible
  optional float  time_to_show = 2;
  // if the message only regards one team
  optional Team   team_color = 3;
}
```

File: `AttentionMessage.proto`

Groups:	S	**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. If the team color is set the message should only be displayed if the color matches the monitored team (or indicate to which team it belongs to).

[6]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;

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

option java_package =
  "org.robocup_logistics.llsf_msgs";
option java_outer_classname =
  "BeaconSignalProtos";

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 jersey number
  required uint32 number = 8;
  // The robot's team name
  required string team_name = 4;
  // The robot's name
  required string peer_name = 5;
  // Team color, teams MUST sent this
  optional Team   team_color = 6;

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

File: `BeaconSignal.proto`

Groups:	P, T	**Period:**	1 sec
Sent by:	robots & refbox	**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. The refbox will set these fields to "LLSF" and "RefBox" respectively. The robots *must* set the team color appropriately, only the refbox itself is allowed to not set it.

The referee box announces its presence on the public channel. The robots must use the team-private channel.

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 position of the robot in the pose field to use it for visualization. Not only does it help the referee, but it will make the games more interesting to watch for visitors.

```
package llsf_msgs;

option java_package =
     "org.robocup_logistics.llsf_msgs";
option java_outer_classname = "VersionProtos";

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`

Groups:	S, P	**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.

```
package llsf_msgs;

option java_package =
    "org.robocup_logistics.llsf_msgs";
option java_outer_classname = "TimeProtos";

// 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`

Groups: — **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";

option java_package =
    "org.robocup_logistics.llsf_msgs";
option java_outer_classname = "Pose2DProtos";

// 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`

Groups: — **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;

option java_package =
    "org.robocup_logistics.llsf_msgs";
option java_outer_classname = "TeamProtos";

// Team identifier.
enum Team {
  CYAN = 0;
  MAGENTA = 1;
}
```

File: `Team.proto`

Groups: — **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 the team color, being either CYAN or MAGENTA (primary field side one or two respectively).

```
package llsf_msgs;

option java_package =
     "org.robocup_logistics.llsf_msgs";
option java_outer_classname = "ProductColorProtos";

enum RingColor {
  RING_BLUE   = 1;
  RING_GREEN  = 2;
  RING_ORANGE = 3;
  RING_YELLOW = 4;
}

enum BaseColor {
  BASE_RED = 1;
  BASE_BLACK = 2;
  BASE_SILVER = 3;
}

enum CapColor {
  CAP_BLACK = 1;
  CAP_GREY  = 2;
}
```

File: `ProductColor.proto`

Groups: — **Period:** —
Sent by: — **Sent to:** —

These enums describe the various colors used in the communication between refbox and robots. The individual colors are prefixed by their respective element.

```
package llsf_msgs;

import "MachineInfo.proto";
import "Pose2D.proto";
import "Team.proto";
import "Zone.proto";

option java_package =
     "org.robocup_logistics.llsf_msgs";
option java_outer_classname =
     "ExplorationInfoProtos";

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 ExplorationZone {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 71;
  }

  // Zone to explore
  required Zone  zone = 2;
  // Team the machine belongs to
  required Team  team_color = 3;
}

message ExplorationInfo {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 72;
  }

  // The signal assignments
  repeated ExplorationSignal  signals = 1;
  // The zones to explore
  repeated ExplorationZone    zones = 2;
}
```

File: `ExplorationInfo.proto`

Groups: P **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 strings to light signal patterns and zones which belong to a specific team. The type is an arbitrary random string that must be returned upon recognizing the defined light pattern. There will be one entry per machine type string to signal light pattern mapping in the signals list. The lights list in the ExplorationSignal sub-message contain one entry per light color, even for signals which are off. The zones field contains a list mentioning all zones assigned to a specific team. A zone may or may not contain a machine.

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 "Team.proto";

option java_package =
    "org.robocup_logistics.llsf_msgs";
option java_outer_classname = "GameInfoProtos";

// Game information for controllers
message GameInfo {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 81;
  }

  // The new desired phase
  repeated string known_teams = 1;
}

// Request setting of a new game phase
message SetTeamName {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 82;
  }

  // Name of the team to set
  required string team_name  = 1;

  // Color of team
  required Team   team_color = 2;
}
```

File:	`GameInfo.proto`		
Groups:	S	**Period:**	start
Sent by:	refbox	**Sent to:**	controller

Additional information about the game. This is sent on initial connection by the refbox to controllers. At the moment it contains a list of known teams that can be set by the controller as playing team.

```
package llsf_msgs;

import "Time.proto";

option java_package =
    "org.robocup_logistics.llsf_msgs";
option java_outer_classname = "GameStateProtos";

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;
    SETUP       = 10;
    EXPLORATION = 20;
    PRODUCTION  = 30;
    POST_GAME   = 40;
  }

  // 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, cyan
  optional uint32 points_cyan = 5;
  // Name of the currently playing team
  optional string team_cyan = 6;

  // Awarded points, magenta
  optional uint32 points_magenta = 8;
  // Name of the currently playing team
  optional string team_magenta = 9;
}

// 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`

Groups:	S, P	**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 "ProductColor.proto";
import "Team.proto";
import "Zone.proto";
import "MachineInstructions.proto";

option java_package =
     "org.robocup_logistics.llsf_msgs";
option java_outer_classname = "MachineInfoProtos";

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, type, and team
  required string    name       =  1;
  optional string    type       =  2;
  optional string    state      =  3;
  optional Team      team_color = 10;

  // Current state of the lights
  repeated LightSpec lights = 7;
  // Optional pose if known to refbox
  optional Pose2D    pose = 8;
  // Which zone the machine is in
  optional Zone      zone = 11;
  // Only set during exploration phase
  // Correctly reported?
  optional bool      correctly_reported = 9;
  // Number of bases loaded
  optional uint32    loaded_with = 13 [default = 0];

  repeated RingColor ring_colors = 14;

  // Instruction information (only clients)
  optional PrepareInstructionBS instruction_bs = 16;
  optional PrepareInstructionDS instruction_ds = 17;
  optional PrepareInstructionRS instruction_rs = 18;
  optional PrepareInstructionCS instruction_cs = 19;
}

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

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

  // Team color (only broadcast)
  optional Team    team_color = 2;
}
```

File: `MachineInfo.proto`

Groups:	S, T	**Period:**	0.25 sec
Sent by:	refbox	**Sent to:**	all

The MachineInfo message is sent periodically to controllers and robots (in production phase only) by the refbox. The message to *controllers* includes static (inputs, output, name, type, pose) and dynamic information (number of additional bases loaded into the machine, light signals, or whether it was correctly reported during the exploration phase) about all machines. It also states the latest prepare instructions received for a machine. The refbox may report only light signals that are on or blinking. Light color that are not mentioned should be considered to be off. It can be used for visually aligning machine representations with the actual position. The message is sent every 0.25 sec.
The message broadcasted to *robots* includes name, type, state, team color, and ring colors (for RS only). It is sent only during the production phase over the team-private channel. At the beginning of the phase, the message is sent at a 0.5 sec period for 15 sec. Afterwards the message is sent every 2 sec.

Both, the MachineInfo and Machine message types feature a team color. The team color on the latter is *always* set. For the MachineInfo, the behavior differs between stream and broadcast messages. For stream messages, it is left out an a common MachineInfo message comprising *all* machines of both teams is sent. For broadcasting, two messages are sent, one for each team. In these MachineInfo messages the team color will be set and only machines matching this color will be included.

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 information allows for cheating.

```
package llsf_msgs;

option java_package =
    "org.robocup_logistics.llsf_msgs";
option java_outer_classname =
    "MachineCommandsProtos";

enum MachineState {
  IDLE      = 1;
  AVAILABLE = 2;
  PROCESSED = 3;
  DELIVERED = 4;
  RETRIEVED = 5;
}

// Load puck into machine area.
message SetMachineState {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 17;
  }

  // Machine to place puck at
  required string machine_name = 1;
  // State to set
  required MachineState state = 2;
}


// Load puck into machine area.
message MachineAddBase {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 18;
  }

  // Machine to place puck at
  required string machine_name = 1;
}
```

File: `MachineCommands.proto`

Groups:	S	**Period:**	event
Sent by:	controller	**Sent to:**	refbox

This protobuf file contains messages for specific instructions to the refbox. The SetMachineState message allow to set an MPS machine state and the MachineAddBase simply increments the additional bases counter by one (RS only).
These commands can be be used to write a game simulation that operates using the refbox.

```
package llsf_msgs;

import "MachineInfo.proto";
import "Team.proto";
import "Zone.proto";

option java_package =
    "org.robocup_logistics.llsf_msgs";
option java_outer_classname = "MachineReportProtos";

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

  // Machine name and recognized type
  // and zone the machine is in
  required string name = 1;
  required string type = 2;
  required Zone   zone = 3;
}

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

  // Team for which the report is sent
  required Team team_color = 2;

  // 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;

  // Team for which the report is sent
  required Team team_color = 2;
}
```

File:	`MachineReport.proto`		
Groups:	T	**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 loss, all machines are reported to the refbox eventually. The message should be send periodically, for example once every two seconds.

The report contains the name, zone, and type string for each recognized machine. The type is the type string announced in the ExplorationInfo message inferred from the recognized light signal.

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 wrong report once received. Every subsequent repetition will be silently ignored. Also ensure that you send reports only for the appropriate team color or your report will be ignored. Sending machine reports for another team's machine with the other team's color is an offense.

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;

import "Team.proto";
import "ProductColor.proto";

option java_package =
    "org.robocup_logistics.llsf_msgs";
option java_outer_classname = "OrderInfoProtos";

message Order {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 42;
  }

  // Ordered product type
  enum Complexity {
    C0 = 0;
    C1 = 1;
    C2 = 2;
    C3 = 3;
  }

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

  required BaseColor    base_color = 3;
  repeated RingColor    ring_colors = 4;
  required CapColor     cap_color = 5;

  // Quantity requested and delivered
  required uint32 quantity_requested = 6;
  required uint32 quantity_delivered_cyan = 7;
  required uint32 quantity_delivered_magenta = 8;

  // Start and end time of the delivery
  // period in seconds of game time
  required uint32 delivery_period_begin = 9;
  required uint32 delivery_period_end   = 10;

  // The gate to deliver to, defaults to any
  // (non-defunct, i.e. non-red light) gate
  required uint32 delivery_gate = 11;

}

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

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


message SetOrderDelivered {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 43;
  }

  required Team team_color = 1;
  required uint32 order_id = 2;
}
```

File:	`OrderInfo.proto`		
Groups:	S, T	**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. Initially, the list is most likely empty. Then, when the lead time of an order has been reached, that is the time in advance with respect to the beginning of the delivery time window, the order is announced by adding it to the orders field. In a message can be orders with an active delivery period or which have already pased or are yet to come.

Orders are sent over the team-private channel and contain only orders for that team.

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

option java_package =
    "org.robocup_logistics.llsf_msgs";
option java_outer_classname = "RobotInfoProtos";

enum RobotState {
  ACTIVE       = 1;
  MAINTENANCE  = 2;
  DISQUALIFIED = 3;
}

message Robot {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 31;
  }

  // Name, team, and number of the robot
  // as it was announced by the robot in
  // the BeaconSignal message.
  required string name = 1;
  required string team = 2;
  required Team   team_color = 12;
  required uint32 number = 7;
  // 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;

  // Pose information (reported by robot)
  optional Pose2D pose = 6;
  // Pose information (reported by vision)
  optional Pose2D vision_pose = 11;

  // Current state of this robot
  optional RobotState state = 8;

  // Time in seconds remaining in current
  // maintenance cycle, negative if overdue
  optional float  maintenance_time_remaining = 9
      [default = 0.0];
  // number of times maintenance
  // has been performed, including current
  optional uint32 maintenance_cycles = 10;
}

message RobotInfo {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 30;
  }

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

File:	`RobotInfo.proto`		
Groups:	S, P	**Period:**	1 sec
Sent by:	refbox	**Sent to:**	controllers

The RobotInfo message is sent periodically by the refbox to connected controllers and over the public broadcast channel. It contains information about all robots known to the refbox gathered from processing BeaconSignal messages. 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. The information is never sent over the broadcast channel.

```
package llsf_msgs;

import "Team.proto";

option java_package =
    "org.robocup_logistics.llsf_msgs";
option java_outer_classname = "RobotCommandsProtos";

// Set a robot's maintenance state
message SetRobotMaintenance {
  enum CompType {
    COMP_ID  = 2000;
    MSG_TYPE = 91;
  }

  // Number of robot to set state
  required uint32 robot_number = 1;
  // True to activate maintenance,
  // false to bring back robot
  required bool   maintenance  = 3;
  // Team the robot belongs to
  required Team   team_color   = 4;
}
```

File:	`RobotCommands.proto`		
Groups:	S	**Period:**	event
Sent by:	controllers	**Sent to:**	refbox

Commands related to robots that controllers can send to the refbox. Currently the only message allows to set the maintenance state of a particular robot of a particular team.

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, Daniel Ewert, Sebastian Reuter, Alexander Ferrein, Sabina Jeschke, and Gerhard Lakemeyer. RoboCup Logistics League Sponsored by Festo: A Competitive Factory Automation Testbed. In *Proc. of RoboCup Symposium 2013*, Eindhoven, Netherlands, 2013.

[2] Tim Niemueller, Gerhard Lakemeyer, Alexander Ferrein, Sebastian Reuter, Daniel Ewert, Sabina Jeschke, Dirk Pensky, and Ulrich Karras. Proposal for Advancements to the LLSF in 2014 and beyond. In *Proc. of ICAR 2013 - 1st Workshop on Developments in RoboCup Leagues*, Montevideo, Uruguay, 2013.

[3] 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.

[4] 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.

[5] Tim Niemueller, Daniel Ewert, Sebastian Reuter, Alexander Ferrein, Sabina Jeschke, and Gerhard Lakemeyer. The Carologistics RoboCup Logistics Team 2013. Technical report, RWTH Aachen University and Aachen University of Applied Sciences, Aachen, Germany, June 2013.

[6] 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.