PSE-C
CH–1015 Lausanne
Switzerland

Tel +41 21 693 83 14
Fax +41 21 693 83 15
www.bluebotics.com

Platform communication interface

RPC over LOS

Revision: 2008-04-09

Table of Contents

Chapter
1

Introduction

Executive summary

1.1 Purpose

This document presents a mechanism for exchanging data through a network connection with all BlueBotics platforms using ANT® technology, and an API using this mechanism for getting information from and controlling a platform.

1.2 Executive summary

This section gives a very short summary of the content of this document:

- The API provides functions for interacting with various subsystems of the platform: localization, motion control, obstacle avoidance, odometry, scan data processing, etc.

- ANT® technology requires an a-priori map of the environment. Maps are simple text files that can be built interactively using a software tool provided by BlueBotics, but can also be generated by other means. The map format is fully documented.

- The API is implemented as a remote procedure call mechanism in a client-server configuration. The platform acts as a server. Clients send procedure calls, and receive either a result or an exception.

- Remote procedure calls and their replies are serialized in a simple, compact binary format called LOS (Lightweight Object Streaming). It allows the efficient transmission of a rich set of data, array and object types. LOS is a proprietary development by BlueBotics. The serialization protocol is fully documented down to individual bits.

- The transport mechanism for remote procedure calls is a TCP connection.

- Two reference implementations of the client side of the RPC over LOS over TCP are provided, one in Python and one in Java. In particular, the Python implementation allows calling any current and future remote procedure in a very natural way.

Chapter

2

RPC interface

Unified programming interface to BlueBotics platforms

2.1 Infrastructure

2.1.1 Procedure name structure

Procedure call names intended for a subsystem are structured into two parts, a subsystem name and a procedure name, separated by a dot. The following sections describe all the available subsystems and the calls they provide.

Infrastructure procedure call names don't have a subsystem part and are therefore only composed of the procedure name.

The naming convention for procedure names is `camelCase`. The naming convention for subsystem names is `CamelCase`.

2.1.2 Procedure call arguments

The arguments to procedure calls are positional, i.e. determined by their position in the list of arguments. Certain calls have optional arguments, which can be left out of the argument list. Optional arguments are always placed at the end of the argument list, and are marked with an asterisk in the following tables.

2.1.3 Procedure call results

Procedure calls always have a single result object. When a call is defined as having no result, the result object is a `Void`. If it is defined as having multiple results, they are packaged as an `Array`.

2.1.4 Authentication

Every connection has an associated authentication level. Connections are initially unauthenticated. Every authentication level gives access to a set of calls, in addition to all the calls inherited from its parent. The authentication level can be changed per connection through the `login()` call.

The following table describes the authentication levels defined on the platform:

Name	Inherits from	Password	Description
`Master`	`User`	{not available}	Master user level, has access to all calls
`User`	`{nobody}`	`none`	Normal user level
`{nobody}`			Unauthenticated level

2.1.5 Infrastructure procedure calls

This section describes a few calls related to the RPC infrastructure that are not part of any subsystem. The `getObjects()` and `inspect()` calls are intended for internal use only and are therefore left undocumented.

2.1.5.1 `getCalls`

This call returns a list of all the procedure names that are available in the current authentication level.

Name	`getCalls`	
Auth level	`{nobody}`	
Arguments		
–		
Results		
`names`	`String[]`	The list of available call names
Exceptions		
–		

2.1.5.2 `getObjects`

This call returns a list of object names corresponding to objects defined on a platform.

Name	`getObjects`	
Auth level	`{nobody}`	
Arguments		
–		
Results		
`names`	`String[]`	The list of available object names
Exceptions		
–		

2.1.5.3 `inspect`

This call allows inspecting the internal state of one or more objects.

Name	`inspect`	
Auth level	`{nobody}`	
Arguments		
`patterns`	`String[]`	A list of patterns identifying the objects to be inspected
Results		
`tree`	`Struct`	The requested values in hierarchical form
Exceptions		
–		

2.1.5.4 `login`

This call changes the credentials on a connection by providing a user name and a password. To de-authenticate a connection, i.e. to return to the lowest authentication level `{nobody}`, use an empty string as `user`.

Name	`login`	
Auth level	`{nobody}`	
Arguments		
`user`	`String`	The authentication level to switch to
`password`	`String`	The password associated with the user
Results		
	`Void`	
Exceptions		
`LoginRefused`		The user / password pair is invalid.

2.2 Localization

The internal localization algorithm periodically extracts features like segments and points from laser scans, matches them to features defined in the map, and corrects the current pose estimate based on this information. It must be switched off if an external localization algorithm is used through `Odometry.update()`.

2.2.1 configure

This call configures the internal localization process. Currently, it only allows stopping and starting the process.

Name	`Localization.configure`	
Auth level	`User`	
Arguments		
`active`	`Boolean`	If true, activate the internal localization system. Otherwise, deactivate it.
Results		
	`Void`	
Exceptions		
–		

2.2.2 snapToNode

This call allows initializing the current pose estimate to a known location by specifying a node defined in the map. It is useful for specifying an initial position after switching on the platform, as well as to re-localize the platform in case the localization algorithm gets lost.

Name	`Localization.snapToNode`	
Auth level	`User`	
Arguments		
`node`	`Int32`	The node of the map to snap to
Results		
	`Void`	
Exceptions		
`Localization.NodeNotFound`		The given node is not in the map.

2.2.3 snapToPose

This call allows initializing the current pose estimate to a known location by specifying a pose in world coordinates. It is useful for specifying an initial position after switching on the platform, as well as to re-localize the platform in case the localization algorithm gets lost.

Name	`Localization.snapToPose`	
Auth level	`User`	
Arguments		
`x`	`Float64`	The X coordinate of the pose to snap to in world coordinates `[m]`
`y`	`Float64`	The Y coordinate of the pose to snap to in world coordinates `[m]`
`theta`	`Float64`	The heading of the pose to snap to in world coordinates `[rad]`
Results		
	`Void`	
Exceptions		
–		

2.3 Map

The map is a container for a-priori information necessary for localization, obstacle avoidance and autonomous navigation. The text format of maps is described in chapter 3.

2.3.1 get

This call returns a text representation of the map currently used on the platform.

Name	`Map.get`	
Auth level	`User`	
Arguments		
–		
Results		
`map`	`String`	A text representation of the current map
Exceptions		
–		

2.3.2 set

This call sets a new map on the platform. In case of a parse error, the line where the error occurred, as well as the actual error, are specified in the exception message.

Name	`Map.set`	
Auth level	`User`	
Arguments		
`map`	`String`	A map in text format
Results		
	`Void`	
Exceptions		
`Map.ParseError`		A parse error occurred while parsing the map.

2.4 Motion

The `Motion` subsystem controls the motion of the platform using various algorithms, and provides status information about the current motion operation.

2.4.1 getSpeed

This call returns the current actual translation and rotation speeds of the platform, and the time of the request.

Name	`Motion.getSpeed`	
Auth level	`User`	
Arguments		
–		
Results		

`result`	`Float64[]`	`result[0]`: the time of the request as an absolute UTC time `[s]`
		`result[1]`: the current platform translation speed `[m/s]`
		`result[2]`: the current platform rotation speed `[rad/s]`

Exceptions		
–		

2.4.2 getStatus

This call returns the state of the motion controller, and the result of the last terminated motion operation, together with the time of the request. `result` is empty while a motion operation is in progress.

Name	`Motion.getStatus`	
Auth level	`User`	
Arguments		
–		
Results		
`time`	`Float64`	The time of the request as an absolute UTC time `[s]`
`state`	`String`	The state of the current motion operation
`result`	`String`	The result of the last terminated motion operation
Exceptions		
–		

When the motion controller is driven, `state` starts either with `Driven` or `Disabled` and the name of the controlling subsystem, possibly followed by additional state information. The following table shows examples of `state` contents in various situations:

Description	state
Motion is disabled	Disabled
Motion controller is ready and not driven	Ready
Motion controller is driven in autonomous mode	Driven.Autonomous
Motion controller is driven in autonomous mode but platform is temporarily blocked	Driven.Autonomous.Blocked
Motion controller is driven in speed control mode, but is disabled (e.g. due to a pushed bumper)	Disabled.SpeedControl

When the motion controller is driven, `result` is empty. When a motion operation has terminated normally (successfully or not), `result` starts with the name of the subsystem that was controlling last, followed by result information The following table shows examples of `result` contents after various events:

Description	result
Motion operation is in progress	
Autonomous motion operation was successful	Autonomous.Success
Autonomous motion operation failed due to a planning error	Autonomous.PlanError
Autonomous motion operation was aborted gracefully	Autonomous.Aborted
Current motion operation was aborted abruptly	Stopped
Speed control watchdog has triggered	TimedOut

These tables are not exhaustive, as every subsystem defines its own states and results. The structure of the strings, however, is always as described above.

2.4.3 moveToNodes

This call starts an autonomous motion operation along the given chain of nodes. Sequential nodes don't necessarily have to be directly connected in the node graph defined in the map. The motion planner sets via nodes when necessary to move from one node to the next along the shortest path.

Name	Motion.moveToNodes	
Auth level	User	
Arguments		
nodes	Int32[]	The list of nodes to be followed
Results		
	Void	
Exceptions		
Motion.Busy		The motion controller is in use by another subsystem

2.4.4 moveToPose

This call starts an autonomous motion operation to the given pose in world coordinates.

Name	`Motion.moveToPose`	
Auth level	`User`	
Arguments		
`x`	`Float64`	The X coordinate of the pose to move to in world coordinates `[m]`
`y`	`Float64`	The Y coordinate of the pose to move to in world coordinates `[m]`
`theta`	`Float64`	The orientation of the pose to move to in world coordinates `[rad]`
Results		
	`Void`	
Exceptions		
`Motion.Busy`		The motion controller is in use by another subsystem

2.4.5 setSpeed

This call sets platform translation and rotation speed target values. The values are used as-is, and no acceleration ramps are applied. If smooth acceleration and deceleration is desired, the ramps must be applied externally.

A one second timeout is activated when controlling the platform using speed commands. If no speed command is received for one second, the platform is stopped abruptly.

Name	`Motion.setSpeed`	
Auth level	`User`	
Arguments		
`sd`	`Float64`	The desired platform translation speed `[m/s]`
`thetad`	`Float64`	The desired platform rotation speed `[rad/s]`
Results		
	`Void`	
Exceptions		
`Motion.Busy`		The motion controller is in use by another subsystem

2.4.6 `stop`

Stop the current motion operation. If `force` is false, the currently driving subsystem is notified and it terminates gracefully. For example, in the case of autonomous navigation, a graceful stop generates a smooth deceleration until the platform is stopped. If `force` is true, the platform is stopped abruptly. This latter case should only be used as a safety stop.

Name	`Motion.stop`	
Auth level	`User`	
Arguments		
`force*`	`Boolean`	If false, stop gracefully. If true, stop abruptly. The default is false.
Results		
	`Void`	
Exceptions		
–		

2.5 ObstacleAvoidance

The ObstacleAvoidance subsystem manages the obstacle avoidance algorithm used during autonomous motion. It is not active in any other motion mode.

2.5.1 configure

This call configures the obstacle avoidance algorithm. Currently, it only allows disabling the use of synchronous scan points for obstacle avoidance. This is useful when obstacle avoidance should run exclusively on externally-provided points.

Name	`ObstacleAvoidance.configure`	
Auth level	`User`	
Arguments		
`disableSync`	`Boolean`	If true, obstacle avoidance does not use synchronous scan points, only asynchronous points. If false, both synchronous and asynchronous scan points are used.
Results		
	`Void`	
Exceptions		
–		

2.6 Odometry

The `Odometry` subsystem manages the current pose estimate and the associated covariances. It integrates wheel encoder information to update the current pose.

The current pose estimate is also updated either by the internal localization algorithm, or through the `Odometry.update()` call.

2.6.1 getPose

This call returns the pose estimate at a given absolute time. If the `time` argument is omitted, the current time is used.

In the current implementation, the odometry history buffer has a length of one second. Therefore, the `time` argument should be at most one second in the past relative to the request time.

Name	`Odometry.getPose`	
Auth level	`User`	
Arguments		
`time*`	`Float64`	The time for which the pose is desired as an absolute UTC time `[s]`. The default is the current time.
Results		
`time`	`Float64`	The time of the pose as an absolute UTC time `[s]`

`pose`	`Float64[]`	The pose at the given time and its associated covariance.
		`pose[0]`: X coordinate of the pose `[m]`
		`pose[1]`: Y coordinate of the pose `[m]`
		`pose[2]`: Orientation of the pose `[rad]`
		`pose[3]`: X coordinate variance `[m`^{2}`]`
		`pose[4]`: Y coordinate variance `[m`^{2}`]`
		`pose[5]`: Orientation variance `[rad`^{2}`]`
		`pose[6]`: X-Y covariance `[m`^{2}`]`
		`pose[7]`: X-orientation covariance `[m*rad]`
		`pose[8]`: Y-orientation covariance `[m*rad]`

Exceptions		
`Odometry.InvalidTime`		The given timestamp cannot be found in the odometry history

2.6.2 update

This call updates the pose estimate at the given absolute time in the past, and re-propagates the motion recorded from the wheel encoders back to the present. This allows accounting for processing time in a localization algorithm. The internal localization algorithm should be stopped before using this call.

In the current implementation, the odometry history buffer has a length of one second. Therefore, the `time` argument should be at most one second in the past relative to the request time.

Name	`Odometry.update`	
Auth level	`User`	
Arguments		
`time`	`Float64`	The time corresponding to the pose as an absolute UTC time `[s]`

`pose`	`Float64[]`	The pose at the given time and its associated covariance.
		`pose[0]`: X coordinate of the pose `[m]`
		`pose[1]`: Y coordinate of the pose `[m]`
		`pose[2]`: Orientation of the pose `[rad]`
		`pose[3]`: X coordinate variance $[m^2]$
		`pose[4]`: Y coordinate variance $[m^2]$
		`pose[5]`: Orientation variance $[rad^2]$
		`pose[6]`: X-Y covariance $[m^2]$
		`pose[7]`: X-orientation covariance `[m*rad]`
		`pose[8]`: Y-orientation covariance `[m*rad]`

Results		
	`Void`	
Exceptions		
`Odometry.InvalidTime`		The given timestamp cannot be found in the odometry history

2.7 Scan

The `Scan` subsystem manages point sets representing the platform's environment. It differentiates between two types of points:

- **Synchronous** points originate from one or more synchronous point producers, typically laser scanners. All scan processing is synchronized to these producers. Only synchronous points are used for the internal localization algorithm.

- **Asynchronous** points originate from non-synchronized point producers. They are used exclusively for obstacle avoidance.

There are several producers of asynchronous points:

- **Virtual walls**: The map allows defining virtual walls, which are converted to equally-spaced asynchronous scan points. This is typically useful for preventing a platform from e.g. falling off stairs, etc.

- **Externally provided points**: The `Scan.addPoints()` call allows providing points from an external source.

The `Scan` subsystem differentiates between fresh points, which are the latest points received from every producer when generating a scan, and memorized points, which originate from earlier scans and are transformed according to the motion of the platform. The capacity for memorized points can be set independently for synchronous and asynchronous points.

Synchronous points are only kept in memory if they don't overlap with the field of view of the fresh scans, and if their age is lower than the configured value. Asynchronous points are only eliminated based on their age. If there is not enough room to keep all memorized points, the oldest points are eliminated.

2.7.1 addPoints

This call adds points to be merged with synchronous and other asynchronous points for obstacle avoidance. The points must be given in platform coordinates.

Name	`Scan.addPoints`	
Auth level	`User`	
Arguments		
`time`	`Float64`	The time at which the points have been sensed as an absolute UTC time `[s]`. If `time = 0`, the current time is used.
`coordinates`	`Float32[]`	The 2D or 3D point coordinates. If 2D coordinates are provided, the `(x, y)` coordinates of point `i` are located at indices `(2*i, 2*i+1)`. If 3D coordinates are provided, the `(x, y, z)` coordinates of point `i` are located at indices `(3*i, 3*i+1, 3*i+2)`.
`is3D*`	`Boolean`	If true, `coordinates` contains 3D coordinates. If false, it contains 2D coordinates. The default is false.
`type*`	`Int8`	The type of the points. The default is `ptExternal`. See 2.7.3 for a list of point types.
Results		
	`Void`	
Exceptions		
–		

2.7.2 configure

This call configures the capacity of the memory for synchronous and asynchronous points, as well as the maximum age of points kept in memory.

Name	`Scan.configure`	
Auth level	`User`	
Arguments		
`syncMemory`	`Int32`	The maximum number of points kept in synchronous point memory `[points]`
`asyncCapacity`	`Int32`	The capacity available for asynchronous points `[points]`
`maxAge`	`Int32`	The maximum age of points in memory `[ms]`
Results		
	`Void`	
Exceptions		
`InvalidParameter.*`		The given parameter value is invalid

2.7.3 get

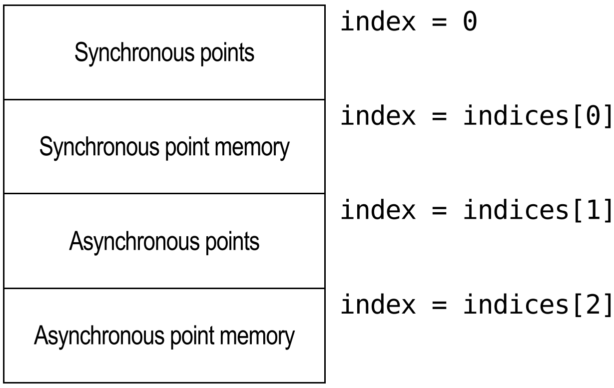
This call retrieves data from the latest merged scan. The points are returned in platform coordinates. The `flags` bit field allows selecting a subset of all available information to limit the necessary bandwidth. The data arrays that are not selected in `flags` are completely absent from the result array.

Name	Scan.get	
Auth level	User	
Arguments		
flags*	Int32	A bit field specifying the data to be returned. The default value enables the flags with an asterisk in the default column below.
Results		
time	Float64	The time at which the scan was taken as an absolute UTC time [s]
pose	Float64[]	The platform pose at the time the scan was taken. pose[0]: X coordinate of the pose [m] pose[1]: Y coordinate of the pose [m] pose[2]: Orientation of the pose [rad]
maxAge	Int32	The maximum age of points in memory [ms]
indices	Int32[]	The start indices of the various zones in the data arrays. indices[0]: start index of synchronous point memory indices[1]: start index of asynchronous points indices[2]: start index of asynchronous point memory
coordinates*	Float32[]	The 2D or 3D point coordinates [m]. If 2D coordinates are requested, the (x, y) coordinates of point i are located at indices (2*i, 2*i+1). If 3D coordinates are requested, the (x, y, z) coordinates of point i are located at indices (3*i, 3*i+1, 3*i+2).
ages*	Int32[]	The age of the points [ms]
intensities*	Int8[]	The measured intensities of the points, as a relative value between 0 and 127.
types*	Int8[]	The type of the points
Exceptions		
ScanMemory.NoScan		No scan is available

The following table describes the bits defined in `flags`:

Bit	Name	Default	Description
0	gfCoordinates	*	2D point coordinates
1	gfCoordinates3D		3D point coordinates
2	gfAges	*	Point ages
3	gfIntensities	*	Point intensities
4	gfTypes		Point types
8	gfSync	*	Synchronous points
9	gfSyncMem	*	Memorized synchronous points
10	gfAsync	*	Asynchronous points
11	gfAsyncMem	*	Memorized asynchronous points

The point data arrays are structured into 4 zones, any of which can be empty depending on the requested data. The following figure shows the structure of the arrays:



The following table describes the point type codes returned in `types`:

Code	**Type**	**Description**
0	`ptSynchronous`	Synchronous point
1	`ptExternal`	Externally-provided point
2	`ptVirtual`	Virtual point, e.g. virtual wall
3	`ptUltrasound`	Point from an ultrasound sensor
4	`ptInfrared`	Point from an infrared sensor

2.8 Test

The `Test` subsystem provides calls intended for testing an implementation of RPC over LOS.

2.8.1 crash

This call crashes the task in which it is executed. This generates a generic `TaskException` with a description of the crash and a stack trace in the associated data.

Name	`Test.crash`	
Auth level	`{nobody}`	
Arguments		
–		
Results		
	`Void`	
Exceptions		
`TaskException`		The call has crashed the current task

2.8.2 nop

This call accepts any number of arguments of any type, and returns a single `Float64` containing the number π.

Name	`Test.nop`	
Auth level	`{nobody}`	
Arguments		
–		
Results		
`pi`	`Float64`	The number π
Exceptions		
–		

2.8.3 throw

This call unconditionally throws an exception with the given name and message. The associated data is a single `Float64` containing the number π.

Name	`Test.throw`	
Auth level	`{nobody}`	
Arguments		
`name`	`String`	The name of the exception
`message`	`String`	The message associated with the exception
Results		
	`Void`	
Exceptions		
`{name}`		An exception with the given name and message. The data associated with the exception is the number π as a `Float64`.

2.9 Watchdog

The `Watchdog` subsystem allows monitoring the network connection between an external client and the platform, and to take safety measures in case the network link is down. The watchdog is initially disabled, and is enabled with the first call to `Watchdog.reset()`.

Currently, the action taken when the watchdog triggers is to stop the platform abruptly.

2.9.1 reset

This call resets the watchdog and sets the trigger time at the given interval in the future. It should be called regularly with a relatively small `interval` so that a broken network link can be detected quickly and acted upon.

Name	`Watchdog.reset`	
Auth level	`User`	
Arguments		
`interval`	`Float64`	The time interval after which the watchdog triggers `[s]`
Results		
	`Void`	
Exceptions		
–		

Chapter
3

Map file format

A-priori map for localization, obstacle avoidance and node graph

3.1 Purpose

The map file is a container for a-priori information about the environment, and is used by several subsystems:

- **Localization**: The internal localization algorithm needs a feature map of the environment for matching features extracted from laser scanner data. The features currently supported are straight segments (i.e. walls) and reflectors, modeled as point features.

- **Autonomous navigation**: The autonomous navigation algorithm can drive along a pre-defined node graph. It automatically selects the shortest valid path from the starting point to the goal. The node graph is a set of nodes connected by a set of unidirectional or bidirectional links.

- **Obstacle avoidance**: The scan data processor takes a list of features from the map, and uses them to create virtual scan points for obstacle avoidance. Currently, only segment features are supported. These segments create a sort of "virtual wall" that the platform treats as if it were real. This allows for example to avoid e.g. stairs or zones with low clearance, which are not visible to the laser scanners.

An initial map is loaded from the boot TFTP server under the name `Init.map2`. If no map is available, a dummy, empty map is set. The current map can be retrieved with `Map.get()`, and a new map can be set at any time with `Map.set()`.

BlueBotics provides an external software tool for interactive construction of maps.

3.2 Structure

The map is a generic container, where objects are contained in "bins". The map parser knows a limited number of directives, which are described in this section. Every directive ends with a tilde character (~).

3.2.1 Bin

The `Bin` directive starts the description of a container bin. It is followed by the type of bin, by the list of objects, and ends with a tilde character (~). The bin types are described in section 3.3. Every object in a bin starts with the object type, followed by a list of type-specific named arguments, and ends with a tilde character (~).

3.2.2 Description

The `Description` directive allows embedding a description in a map. It is currently not used, but can help remembering what a given map is for. It takes a single quoted string as an argument.

3.3 Bin types

This section describes the bin types and their structure.

3.3.1 `Localization.Points`

This bin contains a set of point features that will be matched with reflectors by the localization algorithm. A point is defined by an ID, a position in world coordinates, and the covariance of that position. The covariance indicates how precisely the measurement of the feature has been done. A good first approximation is to set the variances to `0.01` and the covariance to `0.0001`.

The structure of an entry is the following:

```
Point id={id} pos={x} {y} cov={sxx} {syy} {sxy} ~
```

Parameter	Description
`{id}`	A unique identifier for the point. Point features should start numbering at 4000, and should never exceed 4999.
`{x}`	The X coordinate of the point in world coordinates `[m]`.
`{y}`	The Y coordinate of the point in world coordinates `[m]`.
`{sxx}`	The variance of the X coordinate `[m`^{2}`]`.
`{sy}`	The variance of the Y coordinate `[m`^{2}`]`.
`{sxy}`	The X-Y covariance of the coordinates `[m`^{2}`]`.

3.3.2 `Localization.Segments`

This bin contains a set of segment features that will be matched with segments extracted from laser scanner data by the localization algorithm. A segment is defined by an ID, the positions of its endpoints, and the covariance of the endpoint coordinates. The covariance indicates how precisely the measurement of the feature has been done. A good first approximation is to set the variances to `0.01` and the covariance to `0.0001`.

The structure of an entry is the following:

```
Segment id={id} p1={x1} {y1} cov1={sxx1} {syy1} {sxy1}
        p2={x2} {y2} cov2={sxx2} {syy2} {sxy2} ~
```

Parameter	Description
`{id}`	A unique identifier for the point. Segment features should start numbering at 2000, and should never exceed 2999.
`{x1}`	The X coordinate of the first endpoint in world coordinates `[m]`.
`{y1}`	The Y coordinate of the first endpoint in world coordinates `[m]`.
`{sxx1}`	The variance of the X coordinate of the first endpoint `[m`^{2}`]`.
`{sy1}`	The variance of the Y coordinate of the first endpoint `[m`^{2}`]`.
`{sxy1}`	The X-Y covariance of the coordinates of the first endpoint `[m`^{2}`]`.
`{x2}`	The X coordinate of the second endpoint in world coordinates `[m]`.
`{y2}`	The Y coordinate of the second endpoint in world coordinates `[m]`.
`{sxx2}`	The variance of the X coordinate of the second endpoint `[m`^{2}`]`.
`{sy2}`	The variance of the Y coordinate of the second endpoint `[m`^{2}`]`.
`{sxy2}`	The X-Y covariance of the coordinates of the second endpoint `[m`^{2}`]`.

3.3.3 `Navigation.Nodes`

This bin contains the node graph used for autonomous navigation. The graph is composed of a set of nodes, and every node can link to one or more other nodes. The links are directional. To define a bi-directional link, define both unidirectional links.

The structure of an entry is the following:

```
Node id={id} pose={x} {y} {theta} links={id1} {id2} ... ~
```

Parameter	Description
`{id}`	A unique identifier for the node. This identifier must be passed to `Motion.moveToNodes()` for autonomous navigation. Node features should start numbering at 1000, and should never exceed 1999.
`{x}`	The X coordinate of the pose in world coordinates `[m]`.
`{y}`	The Y coordinate of the pose in world coordinates `[m]`.
`{theta}`	The orientation of the pose in world coordinates `[rad]`.
`{id?}`	Identifiers of nodes to which this node links.

This bin also contains a single `Home` object that specifies the node ID where the platform is started. When a map is loaded at boot time, the localization algorithm is initialized to the pose of that node. The home node is not used if a map is sent with `Map.set()`, but the `Home` object is mandatory. The structure of the Home entry is the following:

```
Home node={id} ~
```

Parameter	Description
`{id}`	The identifier of the home node.

The current implementation has a few limitations that will be lifted in the future:

- The node graph must contain at least two, bi-directionally-linked nodes.
- The node graph must be contiguous, and not contain any dead-ends.
- If one of the conditions above is not respected, the map will parse correctly but will lead to undefined behavior of the platform.

3.3.4 `ObstacleAvoidance.VirtualWalls`

This bin contains a set of features that will be used by the scan data processor to generate virtual scan points at regular intervals along the feature. These points are used only for obstacle avoidance, and allow preventing the platform from entering specific zones, for example stairs or low-clearance zones.

The structure of an segment entry is the following:

```
Segment  p1={x1} {y1} p2={x2} {y2} ~
```

Parameter	Description
`{x1}`	The X coordinate of the first endpoint in world coordinates `[m]`.
`{y1}`	The Y coordinate of the first endpoint in world coordinates `[m]`.
`{x2}`	The X coordinate of the second endpoint in world coordinates `[m]`.
`{y2}`	The Y coordinate of the second endpoint in world coordinates `[m]`.

3.4 Example map

The following map is an example map file used in an office scenario. Note that there is no `ObstacleAvoidance.VirtualWalls` bin, as no virtual walls are used in this case.

```
Description "BlueBotics office map" ~

Bin Localization.Segments
    Segment id=2000 p1=0.05 0.1 p2=1.15 0.1 cov1=0.01 0.01 0.0001 cov2=0.01 0.01 0.0001 ~
    Segment id=2005 p1=0.1 3.49 p2=0.1 0.05 cov1=0.01 0.01 0.0001 cov2=0.01 0.01 0.0001 ~
    Segment id=2015 p1=0.6 7.47 p2=0.6 6.27 cov1=0.01 0.01 0.0001 cov2=0.01 0.01 0.0001 ~
    Segment id=2020 p1=1.9 7.59 p2=0.79 7.59 cov1=0.01 0.01 0.0001 cov2=0.01 0.01 0.0001 ~
    Segment id=2060 p1=6.39 3.94 p2=6.38 7.13 cov1=0.01 0.01 0.0001 cov2=0.01 0.01 0.0001 ~
    Segment id=2066 p1=0.23 4.94 p2=0.23 3.87 cov1=0.01 0.01 0.0001 cov2=0.01 0.01 0.0001 ~
~

Bin Localization.Points
    Point id=4020 pos=6.37 7.84 cov=0.0002 0.0002 0.000001 ~
    Point id=4021 pos=3.0404509 4.49361709 cov=0.0002 0.0002 0.000001 ~
    Point id=4022 pos=2.77224399 6.83558353 cov=0.0002 0.0002 0.000001 ~
    Point id=4024 pos=3.86005823 2.85713734 cov=0.0002 0.0002 0.000001 ~
~

Bin Navigation.Nodes
    Node id=1000 pose=3.67892872 3.93833403 3.14159265 links=1005 1025 ~
    Node id=1005 pose=1.46986459 3.98183969 3.14159265 links=1000 1010 1015 ~
    Node id=1010 pose=1.64 6.32 1.57079633 links=1005 1020 ~
    Node id=1015 pose=0.94990973 1.6634537 0.78539816 links=1005 ~
    Node id=1020 pose=2.99 7.45 0.00000001 links=1010 ~
    Node id=1025 pose=4.9 4.47 0.00000001 links=1000 ~
    Home node=1000 ~
~
```

Chapter
4

RPC over LOS over TCP

Remote Procedure Call handshake

4.1 Purpose

This chapter specifies how the remote procedure calls are mapped to LOS objects, and how these objects are transmitted and received using TCP as the transport mechanism. This combination is called RPC over LOS over TCP, or RPC/LOS/TCP in short.

The default port for RPC/LOS/TCP is port 1234.

4.2 Roles

RPC over LOS is purely a client-server protocol. Therefore, the two roles involved are the server and the client.

4.2.1 Server

The server is a purely reactive component, i.e. it never performs any action by itself, except for closing inactive connections. The function of the server can be summarized as follows:

- Start a listener on a pre-defined TCP port and wait for incoming connections.
- While the connection is open, wait for a request object, execute the request, and send back the corresponding reply.
- If no activity is detected on an open connection for a given time (currently 30 seconds), close the connection. This avoids dangling open connections due to e.g. crashed clients.

The server doesn't implement any pipelining in the connection, so there is only ever one request being executed per connection at any given time. It is possible to open several connections to a server, and requests in parallel connections are executed in parallel as well.

4.2.2 Client

The client is the active component. It opens a TCP connection to the server, sends requests, gets replies, and when not needing the server anymore, closes the connection. The function of the client can be summarized as follows:

- Open a TCP connection to the server on the pre-defined port.
- Optionally, authenticate with the server using the `login()` call.
- As long as the connection is open, send a request object, wait for a reply and process the reply.

- Close the TCP connection.

Once a connection is open, any number of transactions can be performed, and it is not necessary to open a new connection for every request.

Authentication is performed per connection. Every connection starts unauthenticated, and a `login()` call lasts until the connection is closed or until the next successful `login()` call.

4.3 Request types

There are two request types: the keepalive and the procedure call.

4.3.1 Keepalive

The server closes any connection that doesn't have any activity for a given time. The keepalive request can be used to artificially generate activity with no other effect than to keep the connection open.

A keepalive request is a single `Void` object. The server's reply is a single `Void` object.

4.3.2 Procedure call

A procedure call is encoded as a `Call` object, containing the name of the call and its arguments.

The server's reply is either a `CallResult` object containing the return value of the call if it was successful, or a `CallException` object containing exception information if the call threw.

A call always returns a result. If no return value was given, a `Void` object is used as the `value` field of the `CallResult` object.

4.4 Implementation comments

4.4.1 Call frequency and latency

The maximum call frequency depends essentially on two parameters: the call execution time and the network latency. By default, the TCP protocol groups small writes into larger packets, and delays their transmission. This is called Nagle's algorithm (see http://en.wikipedia.org/wiki/Nagle's_algorithm). Obviously, this has a negative impact on network latency, and therefore on the maximum possible call frequency.

If a high call frequency is desired, then Nagle's algorithm should be disabled on the client. This is usually done by activating the `TCP_NODELAY` option at the `IPPROTO_TCP` level with a call to `setsockopt()`. In this case, care must be taken that the client transmits requests with a minimal number of `write()` calls, as every write will generate at least one network packet.

Chapter
5

LOS serialization

Lightweight Object Streaming

5.1 Basics

LOS stands for "Lightweight Object Streaming" and is a simple and efficient binary protocol for serializing arbitrary structures composed of objects of a limited set of types into a stream of bytes. This chapter defines the precise encoding of data types and objects into sequences of bytes.

5.2 Simple data types

All objects streamed through LOS are composed of a combination of a few elementary data types. This section describes the streaming of these simple data types into a stream of bytes. The following principles apply to all simple data types:

- All data types are byte-aligned.
- Integer types are serialized in little-endian convention, and are always two's complement signed integers.
- Floating-point types are serialized in IEEE-754 form.
- For all variable-length data types, length information is sent prior to data, to allow for exact memory allocation.

5.2.1 `Boolean`

An individual boolean is serialized as a complete byte where the value is encoded in bit 0, to ensure that data types are always aligned on 8-bit boundaries.

Offset	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Description
0								B	Boolean value

5.2.2 `Int8`

An `Int8` is serialized as a single byte in two's complement form.

Offset	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Description
0	S	b6	b5	b4	b3	b2	b1	b0	Value + sign

5.2.3 Int16

An `Int16` is serialized as 2 bytes, LSB first, in two's complement form.

Offset	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Description
0	b7	b6	b5	b4	b3	b2	b1	b0	Value LSB
1	S	b14	b13	b12	b11	b10	b9	b8	Value MSB + sign

5.2.4 Int32

An `Int32` is sent as 4 bytes, LSB first, in two's complement form.

Offset	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Description
0	b7	b6	b5	b4	b3	b2	b1	b0	Value LSB (0)
1	b15	b14	b13	b12	b11	b10	b9	b8	Value byte 1
2	b23	b22	b21	b20	b19	b18	b17	b16	Value byte 2
3	S	b30	b29	b28	b27	b26	b25	b24	Value MSB (3) + sign

5.2.5 Int64

An `Int64` is serialized as 8 bytes, LSB first, in two's complement form.

Offset	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Description
0	b7	b6	b5	b4	b3	b2	b1	b0	Value LSB (0)
1	b15	b14	b13	b12	b11	b10	b9	b8	Value byte 1
2	b23	b22	b21	b20	b19	b18	b17	b16	Value byte 2
3	b31	b30	b29	b28	b27	b26	b25	b24	Value byte 3
4	b39	b38	b37	b36	b35	b34	b33	b32	Value byte 4
5	b47	b46	b45	b44	b43	b42	b41	b40	Value byte 5
6	b55	b54	b53	b52	b51	b50	b49	b48	Value byte 6
7	S	b62	b61	b60	b59	b58	b57	b56	Value MSB (7) + sign

5.2.6 Float32

A `Float32` is serialized as 4 bytes, LSB first, in IEEE-754 single-precision form.

Offset	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Description
0	f7	f6	f5	f4	f3	f2	f1	f0	Fraction LSB
1	f15	f14	f13	f12	f11	f10	f9	f8	Fraction
2	e0	f22	f21	f20	f19	f18	f17	f16	Fraction MSB + exponent LSB
3	S	e7	e6	e5	e4	e3	e2	e1	Exponent MSB + sign

5.2.7 Float64

A Float64 is serialized as 8 bytes, LSB first, in IEEE-754 double-precision form.

Offset	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Description
0	f7	f6	f5	f4	f3	f2	f1	f0	Fraction LSB
1	f15	f14	f13	f12	f11	f10	f9	f8	Fraction
2	f23	f22	f21	f20	f19	f18	f17	f16	Fraction
3	f31	f30	f29	f28	f27	f26	f25	f24	Fraction
4	f39	f38	f37	f36	f35	f34	f33	f32	Fraction
5	f47	f46	f45	f44	f43	f42	f41	f40	Fraction
6	e3	e2	e1	e0	f51	f50	f49	f48	Fraction MSB + exponent LSB
7	S	e10	e9	e8	e7	e6	e5	e4	Exponent MSB + sign

5.2.8 String

A string is considered to be a sequence of 8-bit characters in ISO-8859-1 encoding. It is serialized as a non-negative Int32 specifying the length of the string, followed by the content characters.

Offset	Content	Description
0	String length LSB (0)	length
1	String length byte 1	
2	String length byte 2	
3	String length MSB (3)	
4	0th char of string	String content
5	1st char of string	
…	…	
3 + length	last char of string	

5.2.9 Array

Array is a heterogeneous array containing an ordered list of objects of arbitrary types. It is serialized as a non-negative Int32 specifying the number of elements in the array, followed by the array items as objects.

Offset	Content	Description
0	Array length LSB (0)	length
1	Array length byte 1	
2	Array length byte 2	
3	Array length MSB (3)	
4	Type code of object 0	Object 0
5	Content of object 0	
o_1	Type code of object 1	Object 1
$o_1 + 1$	Content of object 1	
…	…	…
o_n	Type code of object n	Object n
$o_n + 1$	Content of object n	

5.2.10 `Struct`

`Struct` is a heterogeneous structure mapping strings to objects of arbitrary types. It is serialized as a non-negative `Int32` specifying the number of elements in the structure, followed by `String` / object pairs. The order of elements is undefined.

Offset	Content	Description
0	Struct length LSB (0)	`length`
1	Struct length byte 1	
2	Struct length byte 2	
3	Struct length MSB (3)	
4	Length of key 0	Key 0
8	Content of key 0	
o_{o0}	Type code of object 0	Object 0
$o_{o0} + 1$	Content of object 0	
o_{k1}	Length of key 1	Key 1
$o_{k1} + 4$	Content of key 1	
o_{o1}	Type code of object 1	Object 1
$o_{o1} + 1$	Content of object 1	
…	…	…
o_{kn}	Length of key n	Key n
$o_{kn} + 4$	Content of key n	
o_{on}	Type code of object n	Object n
$o_{n} + 1$	Content of object n	

5.3 Homogeneous array data types

Every simple data type in the set `{Boolean, Int8, Int16, Int32, Int64, Float32, Float64, String}` has an associated homogeneous array type. Except for the `Boolean` array, they are all serialized as a non-negative `Int32` specifying the number of elements in the array, followed by the elements themselves.

Offset	Content	Description
0	Array length LSB (0)	`length`
1	Array length byte 1	
2	Array length byte 2	
3	Array length MSB (3)	
4	Content of element 0	Element 0
o_1	Content of element 1	Element 1
…	…	…
o_n	Content of element n	Element n

5.3.1 `Boolean[]`

A homogeneous array of booleans is serialized as a non-negative `Int32` specifying the number of elements in the array, followed by the elements themselves, packet as 8 elements per byte, starting with the least-significant bit. If the last byte is not completely filled, the last most-significant bits are undefined.

Offset	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Description
0	Array length LSB (0)								`length`
1	Array length byte 1								
2	Array length byte 2								
3	Array length MSB (3)								
4	b7	b6	b5	b4	b3	b2	b1	b0	Elements 0 – 7
5	b15	b14	b13	b12	b11	b10	b9	b8	Elements 8 – 15
…									…
m				b_n	b_{n-1}	b_{n-2}	b_{n-3}	b_{n-4}	Elements (n-4) – n

5.4 Object types

The unit of serialization in the LOS protocol is the `Object`. All objects are serialized as a single-byte type code, followed by the content of the object.

Offset	Field name	Data type	Description
0	type	Int8	Type code
1	...	...	Content

Objects can be arbitrary combinations of simple and compound data types. This section documents the available object types and their type code, and how they are structured.

The following table lists all available object types and their associated type code:

Object type	Type code
Void	0x00
Boolean	0x01
Boolean[]	0x02
Int8	0x03
Int8[]	0x04
Int16	0x05
Int16[]	0x06
Int32	0x07
Int32[]	0x08
Int64	0x09
Int64[]	0x0a
Float32	0x0b
Float32[]	0x0c
Float64	0x0d
Float64[]	0x0e
String	0x0f
String[]	0x10
Array	0x11
Call	0x12
CallResult	0x13
CallException	0x14
Struct	0x15

5.4.1 Void (0x00)

The `Void` object is an empty object, i.e. an object having no content. It is therefore reduced to a single `0x00` byte representing the type code.

Offset	Field name	Data type	Description
0	type	Int8	Type code = 0x00

5.4.2 Objects of simple and homogeneous array data types

All simple data types have a corresponding object where the content is structured as described in 5.2 and 5.3.

5.4.3 `Call (0x12)`

The `Call` object represents a procedure call. It is composed of the name of the procedure and an array of positional arguments.

Offset	Field name	Data type	Description
`0`	`type`	`Int8`	Type code = `0x12`
`1`	`name`	`String`	Procedure name
`n`	`arguments`	`Array`	Positional arguments

5.4.4 `CallResult (0x13)`

The `CallResult` object represents the value returned from a procedure call. It is composed of a single object of arbitrary type. Multiple values can be returned from a procedure call by putting them into an `Array` or a `Struct`. If a procedure call doesn't return any value, a `Void` object is returned.

Offset	Field name	Data type	Description
`0`	`type`	`Int8`	Type code = `0x13`
`1`	`value`	`Object`	Procedure call return value

5.4.5 `CallException (0x14)`

The `CallException` object represents an exception thrown by a procedure call. It is composed of two strings representing the name of the exception and a message describing the exception, followed by an object of arbitrary type giving additional information about the exception.

Offset	Field name	Data type	Description
`0`	`type`	`Int8`	Type code = `0x14`
`1`	`name`	`String`	Exception name
`n`	`message`	`String`	Exception message
`m`	`data`	`Object`	Additional data about exception

Exception names are structured in dotted identifier notation, for example `TypeError`, `InvalidParameter.syncMemory` or `ScanMemory.NoScan`. The exceptions that can be thrown vary from one procedure call to another, and are documented in the procedure call interface.

The exception message is a free-form message explaining the reason for the exception. The `data` field can contain arbitrary call-specific data intended to help finding the cause of the exception, such as a parameter value, a stack trace, etc.

Chapter

6

Reference implementations

Client libraries

6.1 `Los.py`

`Los.py` is the reference implementation of a LOS client in the Python programming language. It requires Python 2.4 or higher.

6.1.1 Installation

On Windows, double-click on the installer and follow the instructions.

On Unix, unpack the source tarball and run the following command as root from the top folder:

```
python setup.py install
```

6.1.2 Package structure

The `Los.py` library is structured as a package `Los` containing the following modules:

Module	Content
`Client`	A class encapsulating a LOS connection which can act as a proxy object for a remote server
`Readers`	A reader capable of reading the simple data types and their corresponding array types from a stream
`Serializers`	The serializer classes for all the LOS object types
`Streams`	A wrapper giving a file-like interface to a socket object
`Types`	The wrapper classes for all the LOS object types
`Writers`	A writer capable of writing the simple data types and their corresponding array types to a stream

6.1.3 Type mapping

The following table shows how the LOS object types are mapped to Python types:

LOS object type	Python type	Inherits from
`Void`	`None`	
`Boolean`	`bool`	
`Boolean[]`	`Los.Types.BooleanArray`	`list`
`Int8`	`Los.Types.Int8`	`int`
`Int8[]`	`Los.Types.Int8Array`	`list`
`Int16`	`Los.Types.Int16`	`int`
`Int16[]`	`Los.Types.Int16Array`	`list`
`Int32`	`Los.Types.Int32`	`int`
`Int32[]`	`Los.Types.Int32Array`	`list`

Int64	Los.Types.Int64	long
Int64[]	Los.Types.Int64Array	list
Float32	Los.Types.Float32	float
Float32[]	Los.Types.Float32Array	list
Float64	float	
Float64[]	Los.Types.Float64Array	list
String	str	
String[]	Los.Types.StringArray	list
Array	list	
Call	Los.Types.Call	object
CallResult	Los.Types.CallResult	object
CallException	Los.Types.CallException	object
Struct	Los.Types.Struct	dict

Only `Void`, `Boolean`, `Float64`, `String` and `Array` map directly to native Python types, and can therefore be specified directly as literals. All other types must be created explicitly in parameter lists. In particular, integer literals don't map to a LOS type, and must therefore be specified e.g. as `Int32(5)`.

6.1.4 Usage

The library is used by instantiating a `Connection` object and calling methods on it.

- Import the relevant modules.

```
from Los.Client import Connection
from Los.Types import *
```

- Instantiate a `Connection` object. `Connection` takes an address parameter, which is a `(host, port)` tuple, and an optional `timeout` parameter for specifying the timeout in seconds on the underlying socket.

```
proxy = Connection(("192.168.8.123", 1234), timeout=2.5)
```

- Call methods on the `Connection` object. The connection can, but doesn't have to, be opened explicitly. The first call will open it implicitly. Keepalives can be sent with the `ping()` method.

```
proxy.open()
proxy.ping()
proxy.login("User", "none")
proxy.Motion.moveToNodes(Int32Array([1000, 1010, 1020]))
while True:
    proxy.Watchdog.reset(1.0)
    (time, state, result) = proxy.Motion.getStatus()
    if result:
        break
    (time, pose) = proxy.Odometry.getPose()
    poses.append((time, pose))
```

- Call exceptions are re-thrown locally by the `Connection` object as a `RemoteException`.

```
try:
    proxy.Motion.moveToNodes(Int32Array([1000, 1010, 1020]))
except RemoteException, e:
    if e.name == "Motion.Busy":
        print "Platform is currently busy"
```

- Close the connection when it isn't needed anymore.

```
proxy.close()
```

6.2 Los.java

`Los.java` is the reference implementation of a LOS client in the Java programming language. It requires a Java 5.0 virtual machine or higher.

6.2.1 Installation

No installation is needed. Just add the provided `.jar` file to the classpath of any projects that need it.

6.2.2 Package structure

The `Los.java` library is structured as a package `com.bluebotics.los` containing the following sub-packages:

Sub-package	Content
`proxy`	A proxy class `AntPlatform` encapsulating a LOS connection and providing methods for all documented calls
`serialization.types`	The serializer classes for all the LOS object types
`serialization.wrappers`	The wrappers classes for object types that cannot be mapped to native type objects
`util`	A few utility classes

6.2.3 Type mapping

The following table shows how the LOS object types are mapped to Java types. The full name of the `wrappers` package is `com.bluebotics.los.serialization.wrappers`.

LOS object type	Java type
`Void`	`wrappers.Void`
`Boolean`	`Boolean`
`Boolean[]`	`boolean[]`
`Int8`	`Byte`
`Int8[]`	`byte[]`
`Int16`	`Short`
`Int16[]`	`short[]`
`Int32`	`Integer`
`Int32[]`	`int[]`
`Int64`	`Long`
`Int64[]`	`long[]`
`Float32`	`Float`
`Float32[]`	`float[]`
`Float64`	`Double`
`Float64[]`	`double[]`
`String`	`String`
`String[]`	`String[]`
`Array`	`Object[]`
`Call`	`wrappers.Call`
`CallResult`	`wrappers.CallResult`
`CallException`	`wrappers.CallException`
`Struct`	`wrappers.Struct`

6.2.4 Usage

The library is used by instantiating a `com.bluebotics.los.proxy.AntPlatform` object and calling methods on it.

- Import the relevant classes.

```
import com.bluebotics.los.RemoteException;
import com.bluebotics.los.proxy.AntPlatform;
import com.bluebotics.los.util.*;
```

- Instantiate an `AntPlatform` object. `AntPlatform` takes the hostname and the port to connect to as parameters. A read timeout in seconds can be set with `setTimeout()`.

```
AntPlatform proxy = new AntPlatform("192.168.8.123", 1234);
proxy.setTimeout(2500);
```

- Call methods on the `AntPlatform` object. The connection can, but doesn't have to, be opened explicitly. The first call will open it implicitly. Keepalives can be sent with the `ping()` method.

```
proxy.open();
proxy.ping();
proxy.login("User", "none");
proxy.Motion_moveToNodes(new int[] {1000, 1010, 1020});
while(true) {
    proxy.Watchdog_reset(1.0);
    Tuple3<Double, String, String> status =
        proxy.Motion_getStatus();
    if(!status.item2.isEmpty())
        break;
    Tuple2<Double, double[]> pose =
        proxy.Odometry_getPose();
    poses.add(pose);
}
```

- Call exceptions are re-thrown locally by the `AntPlatform` object as a `RemoteException`.

```
try {
    proxy.Motion_moveToNodes(new int[] {1000, 1010, 1020});
} catch(RemoteException e) {
    if(e.getName().equals("Motion.Busy"))
        System.err.println("Platform is currently busy");
}
```

- Close the connection when it isn't needed anymore.

```
proxy.close();
```

Chapter

7

Change log

List of changes in this document

7.1 Revision history

7.1.1 2008-04-09

Initial revision, first release to customers.