Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2009-041 September 2, 2009

Lightweight Communications and Marshalling for Low-Latency Interprocess Communicati

Albert Huang, Edwin Olson, and David Moore

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Abstract—

We describe the Lightweight Communications and Marshalling (LCM) library for message passing and data marshalling. The primary goal of LCM is to simplify the development of low-latency message passing systems, targeted at real-time robotics applications. LCM is comprised of several components: a data type specification language, a message passing system, logging/playback tools, and real-time analysis tools.

LCM provides a platform- and language-independent type specification language. These specifications can be compiled into platform and language specific implementations, eliminating the need for users to implement marshalling code while guaranteeing run-time type safety.

Messages can be transmitted between different processes using LCM's message-passing system, which implements a publish/subscribe model. LCM's implementation is notable in providing low-latency messaging and eliminating the need for a central communications "hub". This architecture makes it easy to mix simulated, recorded, and live data sources. A number of logging, playback, and traffic inspection tools simplify common development and debugging tasks.

LCM is targeted at robotics and other real-time systems where low latency is critical; its messaging model permits dropping messages in order to minimize the latency of new messages. In this paper, we explain LCM's design, evaluate its performance, and describe its application to a number of autonomous land, underwater, and aerial robots.

Index Terms—interprocess communication, message passing, robotics middleware, real-time systems

Lightweight Communications and Marshalling for Low-Latency Interprocess Communication

Albert S. Huang[1] Edwin Olson[2] David Moore[1]

[1] Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA

[2] Department of Computer Science and Engineering, University of Michigan, Ann Arbor, MI, USA

E-mail: ashuang@mit.edu, ebolson@umich.edu, dcm@acm.org

1 Introduction

A fundamental software design principle is that of modularity, which promotes maintainability, code re-use, and fault isolation [1], [2]. A large robotic system, for example, can be decomposed into specific tasks such as data acquisition, state estimation, task planning, etc. To accomplish their tasks, modules must exchange information with other modules. With modern operating systems, it is convenient to map individual modules onto software processes that can be on the same or physically separate computational devices. This then transforms the task of information exchange into the well studied problem of interprocess communication.

In this paper, we describe a new message passing system for interprocess communication that is specifically targeted for the development of real-time systems. Our approach is motivated by lessons from modern software practices, and places great emphasis on simplicity and usability from the perspective of a system designer. We call our system Lightweight Communications and Marshalling (LCM) to signify its functionality and its simplicity in both usage and implementation.

The single most notable attribute of mapping modules onto separate processes is that every module receives a separate memory address space. The introduction of this barrier provides a number of benefits; modules can be run on the same or different host devices, started and stopped independently, written in different programming languages and for different operating systems, and catastrophic failure of one module (e.g. a segmentation fault) does not necessarily impact another.

With this independence also comes isolation – sharing information between modules is no longer a trivial task. Module designers must carefully consider what information to share across modules, how to marshal (encode) that information into a message, how to communicate a marshalled message from one module to another, and how to un-marshal (decode) the message once received.

Although a message passing system introduces complexities that must be carefully managed, it also provides opportunities for analysis and introspection that may be invaluable to a developer. In particular, messages may be captured and analyzed by modules specifically designed to aid system development. Such modules might log messages to disk, provide statistics on bandwidth, message rate, etc. If the messages are marshalled according to a formal type system, then a traffic inspection module could automatically decode messages in much the same way a program debugger can automatically decode stack variables of a running application.

LCM provide tools for marshalling, communication, and analysis that are both simple to use and highly efficient. Its overarching design philosophy is to make it easy to accomplish the most common message passing tasks, and possible to accomplish most others. LCM also detects many run-time errors, such as invalidly formatted data and type mismatches.

2 Related Work

The idea of dividing large systems into modules is commonplace in robotics. A variety of message passing systems are currently in use, each with an interesting set of trade-offs. Central design decisions include whether to use a reliable or unreliable data transport and whether to provide an automatic data marshalling system.

Perhaps the best known system is the Player project [3], which provides a client/server model of robot control. A server runs a user-configurable set of drivers in a single process, with each driver in its own thread. Many drivers are included in the Player distribution, each of which has a designated task such as reading camera data or executing wheel speed commands. Users write client programs to interact with drivers via the server, typically by operating on driver "proxy" objects in the client's address space. A proxy object handles communication and data marshalling with the server.

Player provides a programming model familiar to software developers accustomed to traditional application development, and attempts to simplify many aspects of robot development. These efforts, and its diverse set of drivers, factor into its wide ranging success. This architecture has its drawbacks, however. Creating new drivers can be a complex process, and the failure of one driver may impact others, possibly even causing the failure of the entire robot. The monolithic process nature complicates debugging individual components. Player was originally designed for client-driver communication; mechanisms for driver-driver and client-client communication have been added to player, but users looking for a more freely composable system might prefer another framework.

Player uses TCP by default. While TCP itself is reliable and ordered, Player adds a set of "add-replace" rules that cause messages to be dropped if a subscriber cannot keep up with the publisher. Without such a provision, TCP buffers would eventually fill and would cause a slow-down for the publisher.

Internally, Player uses External Data Representation (XDR) [4] for data marshalling. Player's built-in types make use of this facility, and users are encouraged to use XDR themselves. XDR, while serving as the model for our own marshalling system, has a number of drawbacks that we will discuss in Section 3.1.

The Mission Oriented Operating Suite (MOOS) [5] is a message passing system that is particularly popular in the underwater robotics community. It provides a publish/subscribe model in which all communications are routed through a central server, and clients "pull" messages at a fixed rate (typically 10 Hz). MOOS messages are free-form ASCII strings, which has the advantage of making messages easily readable by humans, and the disadvantage of consuming more bandwidth than a binary-encoded message.

The CARMEN robotics package [6] includes the IPC message passing system [7]. IPC uses TCP and a central hub to coordinate communications between modules. By default, the central server routes all communications, though it can also be configured as a "match making" service that is used to facilitate direct client-to-client communications. In both modes, IPC implements a "push" publish/subscribe model. In contrast to a "pull" model, a "push" system sends subscribers messages as soon as possible. IPC provides a facility that partially automates marshalling and un-marshalling of messages, though it requires users to manually keep an ASCII description of the types in sync with a C struct declaration. Users must be sure that the packing and alignment of their structs matches the ASCII description; this is increasingly error prone with machines of varying word sizes.

The Joint Architecture for Unmanned Systems (JAUS) implements a routed message model, in which individual messages are addressed for particular destinations [8]. The primary advantage of JAUS is not the message-passing system itself, but rather that the JAUS specification includes over 200 standardized data types. Modules that implement these standard formats can, in principle, be freely composed together in order to build large systems.

JAUS is designed around a routed message passing system in which each message has a specific destination. An RPC-like mechanism is supported based on "query" and "inform" messages, though components can unilaterally transmit messages (including broadcasts) without a corresponding query. As part of this message-passing framework, JAUS allows a tree of communication nodes. The JAUS specification itself does not specify a transport, but UDP is typically employed. As a result, JAUS does not guarantee message delivery. In comparison to Player and MOOS, the mechanism for message loss is different, but the effects are very similar.

JAUS includes no provision for automatic generation of marshalling code: each data type must be hand coded. Further, the type of a message (a "command code") is encoded in a small 16 bit integer, and the assignment of these values is specified by the JAUS specification. Only a subspace of this set is available for user-defined types, and developers must ensure that these values are assigned in a conflict-free manner.

While JAUS previously supported Java, this support was dropped in OpenJAUS v3.3. Other limitations of the JAUS specification include a message limit of 4096 bytes, and the fact that each node must be manually assigned a globally unique identifier by the system builder.

Robotics Operating System (ROS) is a relatively new framework that aims to provide an entire environment for robotics development. For example, it provides a package management system that automatically handles dependencies. Its messaging subsystem provides a publish/subscribe model and a service-oriented model. A "match maker" process is employed to facilitate client-to-client connections. The message passing interface is generic enough that different transports, including including shared memory, TCP, UDP, and Spread [9], could be used. Initial versions use TCP as the sole message transport.

ROS also provides a data marshalling service based on a type specification language similar to C. As of version 0.6.1, ROS assumes that clients are little-endian. This allows for fast and simple data marshalling on little-endian systems, with the tradeoff that big-endian systems are not supported.

Microsoft's Robotics Development Studio (RDS) [10] differs from most other systems in that instead of providing a publish/subscribe model, it employs a service model. This model can be viewed as a stateful remote procedure call idiom, and is based on the Simple Object Access Protocol (SOAP). Several transport layers are available, ranging from simple memory copies (for services in the same memory space) to XML/RPC for services connected via the Internet.

There are several recurring themes in existing systems. Publish/subscribe models are the most commonly used, with TCP being the most common transport. Most of these systems employ a centralized hub, whether it is used for message routing or merely for "match making". Virtually all of these systems provide a reliable and ordered transport system, though some of the systems provide a UDP transport as a non-standard option. Fig. 1 summarizes some of the key differences.

Service models provide a familiar programming model, but this has its drawbacks. For example, it is typically more difficult to inject previously recorded data into a service-based system. An ability like this is particularly useful when developing perceptual and other data processing algorithms, as the same code can be used to operate on logged data and live data. In a publish/subscribe system, this can be accomplished by simply retransmitting previous messages to clients. Since communications are stateful in a service-oriented system, event injection would require the cooperation of the services themselves.

The systems are widely varied in terms of the support for data marshalling they provide. Binary-formatted messages have the significant advantage of conciseness and are used by most of the systems. Several systems use an XDR-based marshalling system, though some implementations provide only partially automatic code generation. For example, language support is typically limited, and with some systems, the users must manually keep a formatting string in sync with the struct's layout.

Our system, Lightweight Communications and Marshalling (LCM), provides a "push"-based publish/subscribe model. It uses UDP multicast as a low-latency but unreliable transport, thus avoiding the need for a centralized hub. LCM provides tools for generating marshalling code based on a formal type declaration language; this code can be generated for a large number of platforms and operating systems and provides run-time type safety.

Several of the other systems provide an operating "environment", consisting of pre-defined data types, ready-to-use modules, event loops, message-passing systems, visualization and simulation tools, package management, and more. LCM is different in that it is intended to be an "a la carte" message passing system, capable of being integrated into a wide variety of systems.

Finally, the way in which LCM is perhaps most distinctive from other systems is in its emphasis on debugging and analysis. For example, while all systems provide some mechanism for delivering a message from one module to another, few provide a way to easily debug and inspect the actual messages transmitted. Those that do typically do so at the expense of efficiency and performance. LCM provides a tool for deep inspection of all messages passed on the network, requiring minimal developer effort and incurring no performance penalty. This is made possible by design, and allows a system developer to quickly and efficiently identify many bugs and potential sources of failure that are otherwise difficult to detect.

3 Approach

We divide our description of LCM into several sections: type specification, marshalling, communications, and

Package	Communications model	Transport	Marshalling	Messages duplicated	Type safe	Platforms
LCM	Pub./Sub. (push)	UDP Multicast	Automatic	No	Yes	C, Java, Python, MATLAB
Carmen (IPC)	Pub./Sub. (push)	TCP	Partial	Yes	No	C, Java
JAUS	Routed messages	UDP	Manual	Yes	No	C
MOOS	Pub./Sub. (pull)	TCP	Manual (strings)	Yes	No	C++
Player/Stage	Pub./Sub. (pull)	TCP	Automatic for C	Yes	No	C, C++, Java, Tcl, Python
ROS	Pub./Sub. + Service	TCP	Automatic (C++,Py)	Yes	No	C++, Python
Robotics Studio	Service	Multiple	Automatic	Yes	Yes	.NET

Fig. 1. Comparison of commonly used communication packages. The table represents default configurations. Message duplication refers to whether the number of subscribers affects network utilization. LCM and JAUS use UDP-based transports, which provide low-latency messaging at the cost of guaranteed delivery.

tools. Type specification refers to the method and syntax for defining compound data types, the sole means of data interchange between processes using LCM. Marshalling is the process of encoding and decoding such a message into binary data for the communications system which is responsible for actually transmitting it.

3.1 Type Specification

Processes that wish to communicate using LCM must agree in advance on the compound data types that will be used to exchange data. LCM defines a formal type specification language that describes the structure of these types. LCM does not support defining Remote Procedure Calls (RPC), but instead requires applications to communicate by exchanging state in the form of these compound data types. This restriction makes the LCM messages stateless, simplifying other aspects of the system (particularly logging and playback).

The marshalling system, described in Section 3.2 is responsible for encoding this data after being defined by the programmer. It includes a novel scheme for guaranteeing that the applications agree exactly on the type specification used for encoding and decoding. This type-checking system can detect many common types of errors. Like other message passing systems, LCM does not perform any semantic checking on message contents. We note that JAUS provides some trivial semantic checks, in the form of range bounds on numeric values.

LCM defines a type specification language that can be used to create type definitions that are independent of platform and programming language. Each type declaration defines the structure of a message, thus implicitly defining how that message is represented as a byte stream. A code generation tool is then used to automatically generate language-specific bindings that provide representations of the message in a data structure native to the programming language, as well as the marshalling routines. Fig. 2 shows an example of two LCM message type definitions, and excerpts from the C bindings for these types are given in Fig. 3.

```
struct waypoint_t {
    string id;
    float position[2];
}

struct path_t {
    int64_t timestamp;
    int32_t num_waypoints;
    waypoint_t waypoints[num_waypoints];
}
```

Fig. 2. Two example LCM type definitions. The first contains two fields, one of which is a fixed-length array. The second is a compound type, and contains a variable length array of the former in addition to two core data types.

Automatic generation of language-specific source code from a single type definition yields some useful benefits. The most tangible is that a software developer is freed from writing the repetitive and tedious code that allows a module to send and receive messages. The effort required to define a new message and have it immediately ready to use in an application is reduced to be no more than that required to define a native data type or class definition for a programming language.

The LCM type specification was strongly influenced by the External Data Representation (XDR) [4], which is used by Sun Remote Procedure Calls (Sun/RPC) and perhaps most notably, the Network File System (NFS) [11]. Some XDR features are not supported by LCM due to the fact that they are rarely used and are either difficult to implement or invite user error, such as optional data (i.e., support for pointer chasing) and

```
typedef struct _waypoint_t waypoint_t;
struct _waypoint_t {
    char*       id;
    float       position[2];
};

int waypoint_t_encode(void *buf, int offset,
        int buflen, const waypoint_t *p);
int waypoint_t_decode(const void *buf, int offset,
        int buflen, waypoint_t *p);

typedef struct _path_t path_t;
struct _path_t {
    int64_t     timestamp;
    int32_t     num_waypoints;
    waypoint_t *waypoints;
};

int path_t_encode(void *buf, int offset,
        int buflen, const path_t *p);
int path_t_decode(const void *buf, int offset,
        int buflen, path_t *p);
```

Fig. 3. Excerpts from automatically generated C-language bindings for the types defined in Fig. 2. LCM also supports message type bindings for Python, Java, and MATLAB.

unions. For example, XDR can chase pointers in a linked list in order to send the entire list, but a circular reference leads to disaster. We note that unions, part of the XDR specification, are also unsupported by `xdrgen`, the tool employed by IPC.

Other features of XDR create portability issues. For example, LCM only supports signed integer types, since the Java language cannot losslessly represent unsigned types[1]. This least-common denominator approach has ensured first-class support for a variety of languages, including C, Java, MATLAB, and Python. Variants of these languages (such as C++ and Objective C) are also fully compatible. LCM retains XDR's minimal computational requirements, allowing automatically-generated code to run on resource-constrained devices like micro-controllers.

LCM also provides features and other usability improvements over XDR. For example, LCM provides a simple method for declaring the length of a variable-length array; in contrast, the XDR specification does not specify how the length of arrays should be specified, which has led to a variety of incompatible approaches. A second example is LCM's support for namespaces, which make it easier to avoid naming conflicts when sharing code with others. Third, while the XDR specification supports 64 bit integers, this support is omitted from IPC's implementation. IPC also represents booleans as 4 byte quantities, instead of the more efficient single byte quantity used by LCM.

In the following sections, we present more details of the LCM type-specification language. This description is intended to convey the basic structure and feature set of LCM, but is not meant to be comprehensive. The LCM documentation contains a detailed and rigorous treatment.

3.1.1 Data structure syntax

Each LCM data type is placed in a text file by itself and named to match the type. For example, `struct waypoint_t` would be defined in a file `waypoint_t.lcm`. Each struct contains a sequence of fields, where a field has a name and a type. The syntax, shown in Fig. 2, is similar to a C struct.

Primitive data types are supported for each field with the following syntax:

- Signed integers: `int8_t`, `int16_t`, `int32_t`, `int64_t`
- Floating-point values: `float`, `double`
- Strings: `string`
- Boolean: `boolean`

LCM has adopted the C99 "stdint" naming convention for integer types, which explicitly names the size of the data type in bits [12]. Making these sizes explicit (rather than defining an integer in terms of the host's word size, for example), is obviously necessary for cross-platform compatibility. But more importantly, the sizes of data types are a critical aspect of data type design; this explicit notation encourages a deliberate choice.

A member variable of an LCM type can also be another LCM type. In this case, the data from the second type is embedded in the first using the appropriate language-specific idiom. For example, in C, the member is a pointer; in Java, an object reference is used.

Strings also take on the native representation for each language binding. In C, for example, strings are represented by a null-terminated array of `char`; in Java,

1. Early versions of LCM automatically promoted unsigned types to the next largest signed type in Java, but this can create problems. First, programmers may rely on known wrap-around behavior, which this silent type promotion defeats. Second, if the user's Java implementation (perhaps unintentionally) makes use of the additional dynamic range, there is no correct way to encode those values back into the LCM type. Consequently, LCM no longer supports unsigned types. Ultimately, this portability-derived limitation has been a non-issue.

the native `java.lang.String` class is used. Floating point numbers are encoded using the bit format specified by the IEEE 754 standard.

Like XDR, LCM encodes multi-byte values in network (big-endian) byte order. This ensures that messages transmitted between clients running on different architectures can be correctly decoded.

3.1.2 Arrays

LCM supports both fixed and variable length arrays. Fixed length arrays are specified with a numerical value in square brackets after the field name, as in `int array[10]`. In this case, 10 elements will always be encoded.

Variable length arrays are specified by giving the name of another field in square brackets after the field name as in `int array[num_elements]`. The same struct must then contain an integer field with that name (`num_elements` in our example). While LCM and XDR are similar in many respects, they differ in this regard. In fact, the XDR specification [4] does not specify how the length of a variable-length array should be represented, with the result that different implementations have chosen different ways. IPC's implementation is fairly flexible, for example, whereas the utility "xdrgen" imposes several onerous restrictions. In comparison, LCM's method is both intuitive and flexible.

There is also an additional primitive type `byte`, which is useful for specifying arrays of opaque binary data. It is encoded the same as `int8_t`, but will appear in each language binding in a format suitable for representing opaque binary data.

3.1.3 Packages

Some languages such as Java and Python support the concept of namespaces in order to prevent type names from clashing globally. LCM supports this concept via package names, which can be specified at the beginning of an LCM file. For example:

```
package robot;

struct waypoint_t {
    string id;
    float position[2];
}
```

In this case, the type `waypoint_t` now exists in namespace `robot`. In Java, Python, and MATLAB, it would be referenced as `robot.waypoint_t` while in C it would be accessed as `robot_waypoint_t`.

3.1.4 Constants

LCM provides a simple way of declaring constants that can subsequently be used to populate other data fields. Users are free to use these constants in any way they choose, e.g. magic numbers, enumerations, or bitfields.

Constants are strongly typed, and are declared using the `const` keyword:

```
const int32_t YELLOW=1, GOLDENROD=2, CANARY=3;
```

Constants can be specified for all integer and floating point types. String constants are not supported, since strings are not simple value types on many platforms.

3.2 Marshalling

Marshalling refers to the encoding and decoding of structured data into an opaque binary stream that can be transmitted over a network. LCM automatically generates functions for marshalling and unmarshalling of each user-defined data type in each supported language.

The marshalling code generated by LCM automatically ensures that the sender and receiver agree on the format of the message. This mechanism, described in the next section, not only guarantees that the types have the same name (e.g. `waypoint_t`), but also that the type declarations were identical when the sender and receiver were compiled. When a system is in active development, the data types themselves can be in flux: this run-time check detects these sorts of issues.

3.2.1 Type Safety

In order for two modules to successfully communicate, they must agree exactly on how the binary contents of a message are to be interpreted. If the interpretations are different, then the resulting system behavior is typically undefined, and usually unwanted. In some cases, these problems can be obvious and catastrophic: a disagreement in the signedness of a motor control message, for example, could cause the robot to suddenly jump to maximum reverse power when the value transitions from 0x7f to 0x80. In other cases, problems can be more subtle and difficult to diagnose; if two implementations do not agree on the alignment of data fields, the problem may be masked until the value of the data field (or that of an unrelated variable) becomes sufficiently large.

Additionally, as a system evolves, the messages may also change as new information is required and obsolete information is removed. Thus, message interpretation must be synchronized across modules as messages are updated.

3.2.2 Fingerprint

The type checking of LCM types is accomplished by prepending each LCM message with a fingerprint derived from the type definition. The fingerprint is a hash of the member variable names and types. If the LCM type contains member variables that are themselves LCM types, the hash recursively considers those member variables. For example, the fingerprint for `path_t` (see Fig. 2) is a function of the fingerprint of `waypoint_t`.

The fingerprints are prepended to each LCM message, and consume 8 bytes during transmission for each message. Importantly, member variables contained within an LCM declaration do *not* increase the number of bytes of fingerprint data. This is an important refinement: otherwise, the overhead for a `path_t` would be 8 bytes for each `waypoint_t` in the list.

The details of computing a hash function are straightforward and thoroughly documented within the LCM source code distribution, so we omit a detailed description. However, we note that the hash function is not a cryptographic function. The reason is that the hash function for a particular LCM type must be computed at run time: LCM types can be dynamically loaded at run-time (e.g., a dynamically linked library) and it is critical that the hash reflect the type actually being used at run time, and not merely the type that was available at compile time. While these hash values only need to be computed once for each LCM type (and thus do not present a computational burden), requiring an LCM implementation to have access to a cryptographic library is an onerous burden for small embedded platforms.

In the common case, an LCM client knows what type of message is expected on a particular messaging channel. When a message is received by an LCM client, it first reads the fingerprint of the message. If the fingerprint does not match the LCM client's expected fingerprint, a type error is reported.

LCM clients can also build a fingerprint database, allowing them to identify the type of message. This database is particularly easy to construct in Java; using the Java reflection facility, all LCM types in the classpath can be automatically discovered in order to populate this database. This is the technique used by our tool lcm-spy, which allows real-time inspection of LCM traffic.

3.3 Communications

The communications aspect of LCM can be summarized as a publish-subscribe based messaging system that uses UDP multicast as its underlying transport layer. Under the publish-subscribe model, each message is transmitted on a named channel, and modules subscribe to the channels required to complete their designated tasks. It is typically the case (though not enforced by LCM) that all the messages on a particular channel are of a single pre-specified type.

3.3.1 UDP Multicast

In typical publish-subscribe systems, a mediator process is used to maintain a central registry of all publishers, channels, and subscribers. Messages are then either routed through the mediator directly, or the mediator is used to broker point-to-point connections between a publisher and each of its subscribers. In both cases, the number of a message is actually transmitted scales linearly with the number of subscribers. Since many messages will have multiple subscribers[2], this overhead can become substantial.

The approach taken by LCM, in contrast, is simply to broadcast all messages to all clients. A clients simply discard those messages to which it is not subscribed. While at first glance this might appear a wasteful and burdensome requirement, most communication networks make this an efficient operation. Ethernet and the 802.11 wireless standards can be thought of as shared communications media, where a single transmitted packet is received by all devices regardless of destination.

UDP multicast provides a standardized way to leverage this feature, with standardized protocols and programming interfaces implemented on every major operating system. Additionally, implementations are generally highly optimized and efficient as a direct result of being tightly coupled with the operating system's IP network stack. For these reasons, LCM bases its communications directly on UDP multicast.

LCM is conceptually divided into networks, where every client on an LCM network receives every message transmitted from other members of that network. An LCM network is directly mapped to a UDP multicast group and port, and so all LCM clients operating on a given network transmit packets to the same multicast group address and port number. A maximum LCM message size of 4 GB is achieved via a simple fragmentation and reassembly protocol.

The time-to-live (TTL) parameter of multicast packets is used to control the scope of a network, and is most

2. An example of a widely-subscribed message is the message containing the robot's position. Other examples include logging or data visualization applications, which subscribe to most or all messages.

commonly set to 0 or 1. In the case where every software module is hosted on the same computational device, then the TTL is set to 0. The result is that messages are transmitted to every module on the device, but are not transmitted onto the physical network. When devices are connected via Ethernet, the TTL is set to 1.

Since LCM uses a multicast mechanism, it does not require a centralized hub for either relaying messages or for "match making". There is also no need for a publisher to keep track of its subscribers, which can be error-prone if subscribers periodically start and stop (either by design or due to application errors).

While UDP Multicast was designed to be able to distribute traffic over more complicated networks (including the Internet), LCM is intended for tightly-coupled systems connected via a private ethernet.

3.3.2 *Delivery Semantics*

LCM provides a best-effort packet delivery mechanism; as a result, LCM messages can be lost or arrive out of order. The use of an "unreliable" transport makes LCM comparable not only to JAUS (which uses UDP), but also MOOS and Player. While MOOS and Player both use a reliable transport (TCP), they allow messages to be lost in order to manage situations in which the subscriber cannot keep up with a publisher.

However, systems that build upon reliable transports (like TCP) can encounter additional latency due to the transport. These transports must include mechanisms for loss detection and packet retransmission. This introduces latency and may also delay future messages. A system that relies on low-latency communications and has significant real-time constraints, such as a robot, may in some cases prefer that a lost packet simply be dropped so that it does not delay future messages. For example, if a wheel encoder reading or a vehicle pose estimate is lost in transmission, it may be more desirable to simply wait for the next update.

A second reason to forgo reliable delivery semantics is that on a distributed system, the loss of a message may not simply be the result of a transient network failure. Instead, it may be symptomatic of a module failure or even the failure of the network itself. These failures are still possible on a system that provides reliable delivery semantics, and a well designed system must be able to handle them robustly.

Transient network failures leading to dropped packets are often used as a reason to include packet retransmission features. However, we note that a properly cabled 1000Base-T Ethernet can typically expect a bit-error rate of at most 10^{-12} [13]. In general, UDP packet loss over wired networks is much more commonly a result of overflowing packet buffers rather than physically corrupted packets.

In short, LCM gives strong preference to the expedient delivery of recent messages, with the notion that the additional latency and complexity introduced by retransmission of lost packets does not justify delaying newly transmitted messages. To this end, it also does not attempt to globally or causally order messages. These semantics may still be implemented on top of the LCM message passing service, but in practice we have found the default semantics to be sufficient for the vast majority of messages passed between our robotic software modules.

3.4 Tools

To assist development of modular software systems, LCM provides several tools useful for logging, replaying, and inspecting traffic. The logging tools are similar to those found in many interprocess communications systems, and allow LCM traffic to be recorded to a file for playback or analysis at a later point in time. The inspection tools allow real-time decoding and display of LCM traffic with no system overhead (such as additional network bandwidth) or developer effort. Together, these tools allow a developer to rapidly and efficiently analyze the behavior and performance of an LCM system.

For maximum portability, all tools with a graphical user interface are written in Java.

3.4.1 *Logging and Playback*

The logging tool provided by LCM is simple while providing intuitive access to more complex options. It subscribes to all channels, and writes every message received to disk together with the time of receipt and the channel on which it was received.

A log playback tool allows a user to load a log file from disk, and then retransmit the logged data as live traffic. This is often useful for reviewing data collected on a previous mission, or for post-processing data. Playback can be sped up or slowed down for brief review or careful analysis of a rapid sequence of messages, and a slider is provided for rapidly finding an area of interest in the log file. We note that, unlike some other systems, a log player is not special in any way: it is simply another LCM client that transmits messages. LCM is very flexible in mixing data sources: it is possible, for example, to run a logger and a log player simultaneously, or to even play back two logs simultaneously.

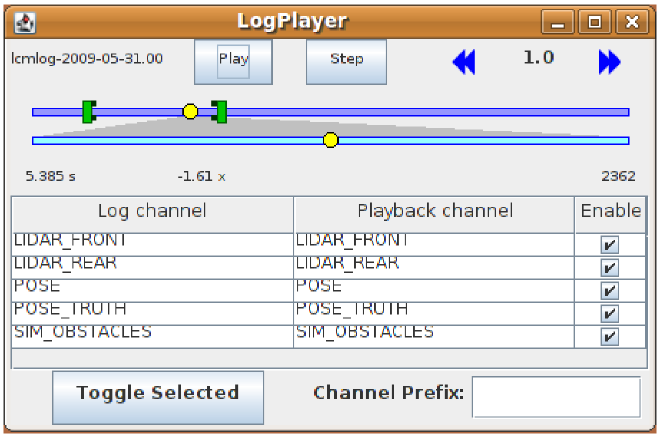


Fig. 4. LCM log player screenshot. The user can adjust playback speed and log position, in addition to adjusting which logged data is played back (changing the channel name if desired). In this image, the user has added “repeats” around an early portion of the log.

Individual channels can be selectively disabled during playback. This feature is often useful when developing estimation and inference algorithms. For example, the logged results of an obstacle detector can be disabled during playback while the original sensor data is replayed. Then a new obstacle detector can operate on the logged data as though it were live traffic.

The log player tool also supports bookmarks and looping functionality. The user can add “repeat signs”, similar to those used in musical notation, which will cause the log player to endlessly repeat that subsection of the log. This is useful when trying to understand an anomalous sensor reading, for example. The positions of bookmarks and repeats are automatically saved to disk and can be shared between different users. A screenshot of the logplayer is shown in Fig. 5.

Console versions of the logger and logplayer are also available, when the features of the graphical versions are not needed.

3.4.2 Spy

Although the primary purpose of an LCM type fingerprint is to detect runtime type mismatches, it also serves another useful purpose. If a database of every LCM type used on a system is assembled, then an arbitrary fingerprint also serves as a type identifier with very high probability[3] The lcm-spy tool is designed to leverage this attribute, and is able to analyze, decode, and display live traffic automatically with virtually no programmer effort. Fig. 5 shows a screenshot of lcm-spy inspecting traffic.

lcm-spy is implemented in Java, and requires only that the classpath contain the automatically generated Java versions of each type. Using the reflection features of Java, it searches the classpath for classes representing LCM types, building a mapping of fingerprints to LCM types. Because each field of an LCM message is strongly typed, lcm-spy is able to automatically determine a suitable way to decode and display each field of a message as it is received.

User-provided data rendering “plugins” are also supported for custom display of individual message types. Commonly used plugins include a graphical display for laser data and an image renderer for camera data.

In addition to its message decoding capabilities, lcm-spy also provides a summary of messages transmitted on all channels along with basic statistics such as message rate, number of messages counted, and bandwidth utilized. Together, these features provide a unique and critically useful view into the state of messages on an LCM network.

When used in practice, lcm-spy allows developers to quickly identify many of the most common failures. During module development, it can help verify that a module is producing messages on the correct channels at the expected rate and bandwidth. It can also be used to inspect arbitrary messages to check the values of any field, useful for tracking down bugs and validating the correct operation of a module.

In our experience, lcm-spy is a critically important tool, on par with program debuggers and profilers. Whereas a debugger is useful for inspecting the internal state of a module, lcm-spy has become invaluable for inspecting the state of the messages passed between modules. Because it passively observes and analyzes traffic, lcm-spy can provide this insight with absolutely no impact on the system performance.

4 PERFORMANCE

One way to measure the interprocess communication performance of LCM is by examining its bandwidth, latency, and message loss rate under various conditions.

3. The fingerprints of each LCM type are represented as 64 bit integers, providing theoretical collision resistance for 2^{32} different types. While LCM’s non-cryptographic hash function degrades this figure, the probability of collisions is vanishingly small for the few hundred message types that a large system might employ.

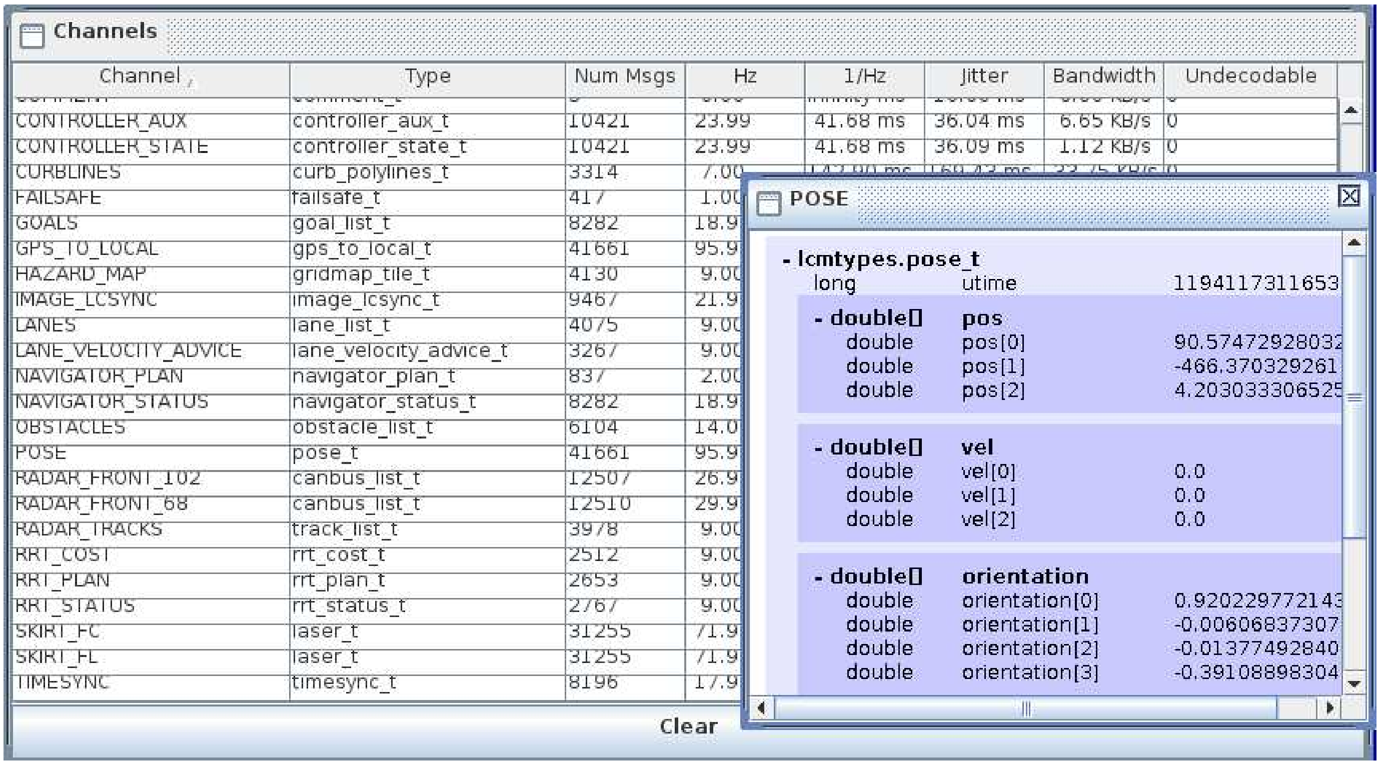


Fig. 5. LCM-spy screenshot. lcm-spy is a traffic inspection tool that is able to automatically decode and display LCM messages in real-time. It requires only that the automatically generated Java code for the LCM types be accessible in the class path; no additional developer effort is required.

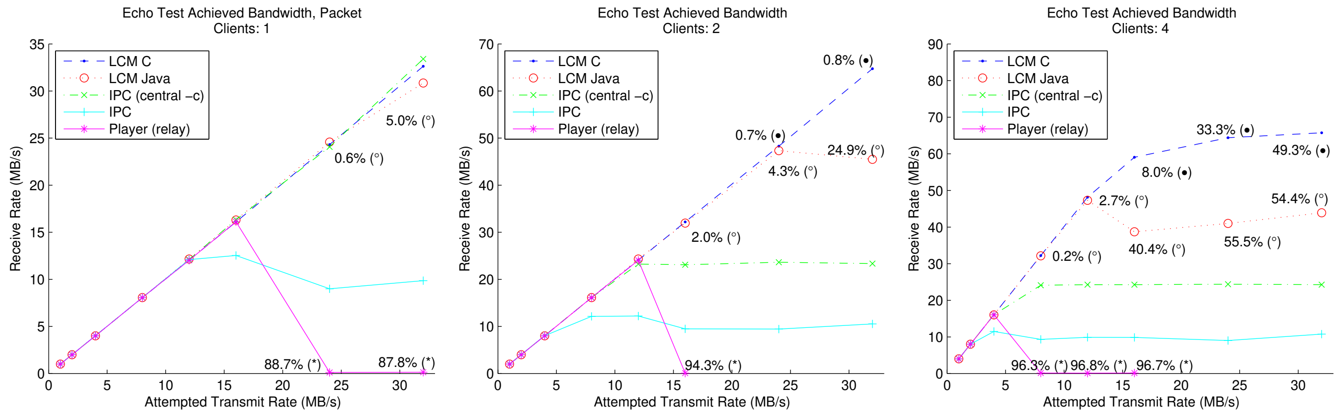
Figures 6 and 7 show the results of a messaging test comparing the C and Java implementations of LCM with IPC and Player.

In this test, a source node transmits fixed-size messages at various rates. One, two, or four client nodes subscribe to these messages and immediately retransmit (echo) them once received. The source node measures how many messages are successfully echoed, the round-trip time for each echoed message, and the bandwidth consumed by the original transmission and the echoes. For this test, IPC is run in two modes, one in which the central dispatching server relays all data (the IPC default), and another in which the central server acts merely as a "match maker" facilitating peer-to-peer connections (`central -c`). To improve performance, operating system send and receive windows were increased to 2MB, and TCP buffers were increased to 4MB. These settings also affect the performance of Player and LCM.

The Player test is implemented by using the "relay" driver to transmit messages between multiple processes connected to the player server. This is not a typical configuration, as Player is more conducive to monolithic process design. However, we believe it to be a reasonable choice if one were to use the Player framework for message passing between arbitrary client processes.

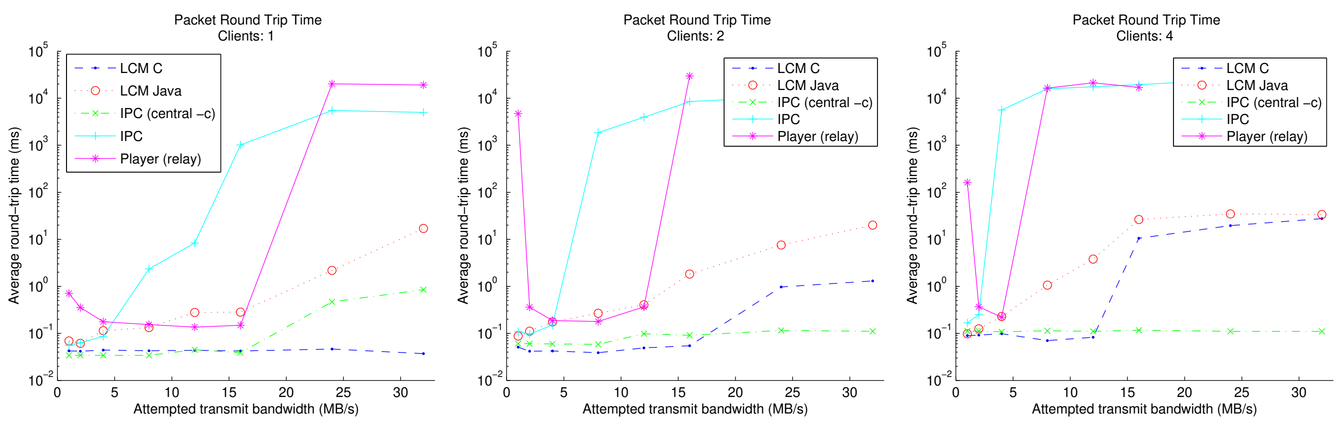
To collect each data point, the sender transmits 100MB of data split into 800 byte messages at a fixed rate determined by a target transmission bandwidth. We chose 800 byte messages because they are roughly the size of a lidar scan produced by a SICK sensor, and are one of the most commonly used data types. Figures 6 and 7 show the results for tests conducted on a quad-core workstation running Ubuntu 8.04. Hosting each process on separate identical workstations connected via 1000Base-T Ethernet yielded similar results. In some cases, the actual message transmission rate does not match the target transmission rate due to transport and software limitations. In a real system, these limitations could result in degraded performance.

From these figures, we can see that LCM scales with both the amount of traffic to be sent and the number of subscribers. As ideal network capacities are reached, such as in Fig. 6c, LCM minimizes latency and maintains high bandwidth by dropping messages. The LCM Java implementation performs comparably to the C implementation, and responds to computational limits



(a) (b) (c)

Fig. 6. Bandwidth results from an echo test with 1, 2, and 4 clients. Each client echoes messages transmitted by a single sender, and the bandwidth of successful echoes as detected by the original sender are shown. Message loss rates (messages with no received echo) are shown in parentheses when nonzero.



(a) (b) (c)

Fig. 7. Mean message round-trip times for the echo test in Fig. 6 with 1, 2, and 4 clients. Average round-trip times are shown on a log scale; these times do not reflect lost packets (which essentially have an infinite round-trip time). Both LCM implementations offer low latency. While IPC (using central -c) also appears to provide low latency, it is critical to notice that IPC's achieved bandwidth fell short of the target bandwidth (see Fig. 6).

of the virtual machine by dropping more messages.

IPC also performs well in the case of one subscriber. However, it does not scale as well to multiple subscribers due to its need to transmit multiple copies of a message. Using the match-making service of IPC (`central -c`) improves bandwidth and reduces latency, but ultimately has the same difficulties. We note that although IPC with peer-to-peer connections maintains low latency as the attempted transmission rate is increased in Figs. 7b and 7c, the actual bandwidth achieved does not increase due to link saturation from duplicated messages. For example, with four clients echoing messages, IPC is unable to transmit faster than 11 MB/s, as the bandwidth consumed by the quadruplicate transmission and the echoes saturates the link capacity.

Our initial experiments with IPC, using the distribution in Carmen 0.7.4-beta, produced much worse results. We determined that this was due to coarse-grained timing functions that degraded performance when packet rates exceed approximately 1 kHz. We were able to improve this performance by exporting higher-resolution versions of those functions; this improved data is used in this

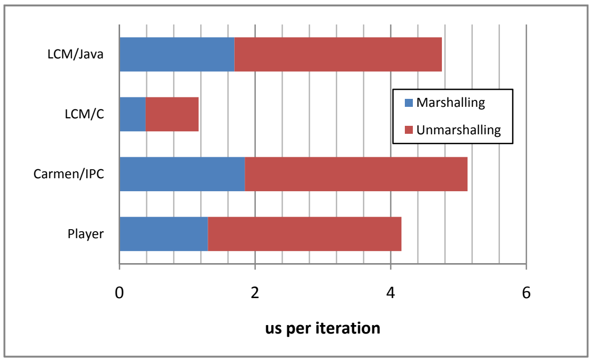


Fig. 8. Marshalling performance. Performance (in microseconds per iteration) is shown for four different marshalling implementations. The LCM implementation in C is more than four times faster than the next fastest marshaller. Notably, the LCM Java implementation, despite being written in pure Java, is comparable to the Carmen and Player implementations.

```
struct laser_t
{
  int64_t utime;
  int32_t nranges;
  float   ranges[nranges];
  int32_t nintensitiese;
  float   intensitiese[nintensitiese];
  float   rad0;
  float   radstep;
}
```

Fig. 9. The LCM type definition used in the marshalling test. Messages were populated with 180 range measurements and no intensity measurements.

paper.

Player does not perform as well, largely because it has not been optimized for client-client communication. For low transmission rates, it scales well, but its performance drops off precipitously as bandwidth is increased.

4.1 Marshalling Performance

In addition to performance of the communications system, we are also interested in the computational efficiency of the LCM marshalling and unmarshalling implementations. Performance of a marshalling system is a function of the complexity and nature of the message, and the presence of arrays, nested types, strings, and other fields are all factors. We have chosen to compare the performance of LCM, IPC, and Player[4] on a commonly used message – one containing data from a single scan of a planar laser range finder with 180 range measurements and no intensity measurements. Fig. 9 shows the LCM type definition. Similar message types were defined for IPC and Player.

In this test, we estimate the amount of time each implementation spends encoding and decoding a single message by measuring the time taken to encode and decode 10^6 messages, and averaging the result. Estimates were taken 10 times and then averaged (warm-up periods for Java were excluded from this average). Timings are shown in Figure 8. The LCM C implementation was the fastest, averaging 0.38 μs per encode, and 0.40 μs per decode. Player/XDR was the second fastest, with the LCM Java implementation and Carmen/IPC close behind.

4. Since Player uses XDR internally, this can also be treated as a comparison against XDR.

5 Case Studies

Since its development, LCM has been used as the primary communications system for a number of robotics research platforms operating in real environments. In this section, we describe several of these platforms and how LCM was applied to the development of each system.

5.1 Urban Challenge

The 2007 DARPA Urban Challenge was an autonomous vehicle race designed to stimulate research and public interest in autonomous land vehicle technology. Vehicles were required to safely navigate a 90 km race course through a simulated urban environment in the presence of both robotic- and human-driven vehicles. LCM served as the communications backbone for both the Ford/IVS vehicle and the MIT vehicle. The MIT vehicle was one of six to complete the race [14].

The MIT vehicle was equipped with a cluster of 10 quad-core workstations connected via a switched local area network. At any given point in time, 70 separate modules were in active operation. The average bandwidth used by the entire LCM network was 16.6 MB/s, with an average transmission rate of 6,775 messages/s. Table 1 provides a detailed breakdown of messaging

Module Category	Total msg/s	Total kB/s
SICK LIDAR	887.1	668.0
Velodyne LIDAR	2.562.2	3,141.2
GPS/INS	774.5	123.5
Radar	443.9	310.7
Camera	163.3	10,082.3
State Estimation	288.0	1,372.8
Planning and Control	175.6	500.6
Debugging	1.146.7	358.2
Other	333.6	25.4
Total	6,774.9	16,582.7

TABLE 1
LCM traffic summary on MIT Urban Challenge vehicle

rates and the bandwidth used by various modules on the vehicle.

Messages transmitted on the network ranged from very small updates such as time synchronization packets to camera images and obstacle maps which were often several hundred kilobytes or several megabytes in size. Some messages, such as the pose estimates, were subscribed to by virtually every module on the network, while others had only one or two subscribers.

Throughout the development process, almost 100 different message types were used, many of which changed frequently as the capabilities and requirements of each module evolved. Software development was distributed across many people working from different locations, and the LCM type definitions became a convenient place for developers to agree on how modules would interface.

Because language-specific bindings could be generated automatically from LCM type definitions, modifying messages and the modules that used them to add or remove fields could often be accomplished in the span of minutes. Additionally, the runtime type checking of LCM fingerprints provided a fast and reliable way to detect modules that had not been recompiled to use the newly modified form of a message.

5.2 Land and Underwater Vehicles

Since the Urban Challenge, LCM has been applied to a number of other autonomous land vehicle research platforms such as an autonomous forklift and small indoor wheeled robots. In many cases, modules used in one vehicle were easily transitioned to other vehicles by ensuring that the LCM messages they needed for correct operation were present on the target robot.

In addition to assisting development of robotic ground vehicles, LCM has also been successfully applied to a number of autonomous underwater vehicles (AUV) at MIT, University of Michigan [15], and the Woods Hole Oceanographic Institute. Development of these vehicles is often characterized by the risk of failure; since underwater robots are typically designed to operate in environments not easily reached by humans, a failed field operation sometimes results in an unrecoverable vehicle.

The University of Michigan Perceptual Robotics Lab has developed a set of AUV platforms for researching underwater robotic mapping, cooperative multi-vehicle navigation, and perception-driven control [15]. Each vehicle contains on-board sensors, thrusters, and a computer for data processing and vehicle control. Despite a vastly different application domain from the Urban Challenge and a simpler network topology, the software engineering principles remain identical, and LCM has proved just as useful. New message types are easily defined as needed, and software modules are adapted to operate in different domains.

5.3 Autonomous Indoor Flight

Indoor environments are among the most difficult regimes for aircraft operation. Recent advances in battery power density, processor improvements, and lightweight materials have opened the door for significant progress in autonomous indoor aerial vehicles. LCM is deployed on one such system actively being used for algorithmic research in planning under uncertainty [16].

The vehicle is a custom-built quadrotor helicopter. On-board sensing consists of an inertial measurement system, three cameras, and a Hokuyo UTM planar laser range scanner. Most processing is done on-board using an Intel Atom processor. As before, modules are divided into separate processes exchanging information via LCM messages.

Unlike a passenger vehicle, the quadrotor is not large enough to carry a human developer or a computer display. A more powerful workstation is used for real-time analysis and debugging. Communication between the quadrotor and the workstation is maintained via an 802.11n wireless connection, although the LCM UDP multicast protocol is not used to transmit messages over this link due to the relatively high rate of packet loss. Instead, inter-host communication is achieved by encapsulating LCM messages in a transport more suitable for wireless transmission.

In this case, LCM messages on the quadrotor are transmitted to a more powerful workstation by means of a wireless UDP tunnel. Packets passing through the tunnel are encoded with a low-density parity check (LDPC) forward error correcting (FEC) codec to improve the probability of successful transmission [17]. A TCP tunnel was experimented with, but the FEC strategy provides higher bandwidth and lower latency with acceptable packet loss. Additional LCM modules are run on the workstation for data visualization, logging, and computationally intensive tasks such as image processing and laser mapping. The tunnel effectively serves as a bridge linking two separate LCM networks.

6 Conclusion

In this paper, we have presented LCM and its design principles. LCM is driven by an emphasis on simplicity and a focus on the entire development process of a robotic software system. In addition to achieving high performance, LCM also provides tools for traffic inspection and analysis that give a developer powerful and convenient insight into the state of the robotic system.

The LCM type specification language is designed to allow flexible and intuitive descriptions of a wide class of data structures. Type fingerprints allow for runtime type checking and identification, and automatically generated language bindings result in a simple and consistent API for manipulating messages and the data they represent. Native support for multiple programming languages allows developers to choose the environment most suitable for the task at hand.

The communications aspect of LCM is designed around the needs of a robotic system and takes advantage of commonly encountered configurations. Low latency is favored over guaranteed delivery semantics, and UDP multicast is used to provide high bandwidth and scalability.

To date, LCM has been successfully deployed as the core communications infrastructure on a number of demanding robotic systems on land, water, and air. These include the MIT Urban Challenge vehicle, a quadrotor helicopter, several autonomous underwater vehicles, and a robotic forklift. In each of these scenarios, the simplicity and versatility of LCM allowed for rapid development of complex software systems. The modular nature of these systems has allowed for significant code re-usability and application of modules developed on one system to another.

LCM is distributed at http://lcm.googlecode.com.

References

[1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972. 1

[2] B. Meyer, *Object-Oriented Software Construction*. Prentice Hall PTR, March 2000. 1

[3] T. H. Collet, B. A. MacDonald, and B. P. Gerkey, "Player 2.0: Toward a practical robot programming framework," in *Proceedings of the Australasian Conference on Robotics and Automation*, Sydney, Australia, Dec. 2005. 2

[4] R. Srinivasan, "XDR: external data representation standard," http://www.rfc-editor.org/rfc/rfc1832.txt, Internet Engineering Task Force, RFC 1832, Aug. 1995. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1832.txt 2, 3.1, 3.1.2

[5] P. M. Newman, "MOOS - mission orientated operating suite," Massachusetts Institute of Technology, Tech. Rep. 2299/08, 2008. 2

[6] M. Montemerlo, N. Roy, and S. Thrun, "Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (carmen) toolkit," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*, vol. 3, Las Vegas, NV, USA, October 2003, pp. 2436–2441. 2

[7] R. Simmons and D. James, *Inter-Process Communication*, August 2001. 2

[8] J. W. Group, *The Joint Architecture for Unmanned Systems: Reference Architecture Specification*, June 2007. 2

[9] Y. Amir and C. Danilov, "The Spread wide area group communication system," Center for Networking and Distributed Systems, Johns Hopkins University, Tech. Rep. Technical Report CNDS-94-4, 2000. 2

[10] J. Jackson, "Microsoft robotics studio: A technical introduction," *Robotics & Automation Magazine, IEEE*, vol. 14, no. 4, pp. 82–87, 2007. [Online]. Available: http://dx.doi.org/10.1109/M-RA.2007.905745 2

[11] Sun Microsystems, "NFS: network file system protocol specification," http://www.rfc-editor.org/rfc/rfc1094.txt, Internet Engineering Task Force, RFC 1094, Mar. 1989. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1094.txt 3.1

[12] ISO, "ISO/IEC 9899:1999 Programming Languages - C," 1999. [Online]. Available: http://www.open-std.org/JTC1/SC22/wg14/www/docs/n1124.pdf 3.1.1

[13] C. DeSanti, "1000base-T and Fibre Channel," Tech. Rep. Cisco T11/05-311v0, November 2005. 3.3.2

[14] J. Leonard *et al.*, "A perception-driven autonomous vehicle," *Journal of Field Robotics*, vol. 25, no. 10, pp. 727–774, Oct 2008. 5.1

[15] H. C. Brown, A. Kim, and R. M. Eustice, "An overview of autonomous underwater vehicle research and testbed at PeRL," *Marine Technology Society Journal*, 2009. 5.2

[16] M. Achtelik, A. Bachrach, R. He, S. Prentice, and N. Roy, "Stereo vision and laser odometry for autonomous helicopters in GPS-denied indoor environments," G. R. Gerhart, D. W. Gage, and C. M. Shoemaker, Eds., vol. 7332, no. 1. SPIE, 2009, p. 733219. [Online]. Available: http://link.aip.org/link/?PSI/7332/733219/1 5.3

[17] C. Neumann, V. Roca, M. Cunche, and J. Labouré, "An Open Source LDPC Large Block FEC Codec," http://planete-bcast.inrialpes.fr, 2009. [Online]. Available: http://planete-bcast.inrialpes.fr 5.3