LCM: Lightweight Communications and Marshalling

Albert S. Huang, Edwin Olson, David C. Moore

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, especially for real-time robotics research applications.**

Messages can be transmitted between different processes using LCM's publish/subscribe message-passing system. A platform- and language-independent type specification language separates message description from implementation. Message specifications are automatically compiled into language-specific bindings, eliminating the need for users to implement marshalling code while guaranteeing run-time type safety.

LCM is notable in providing a real-time deep traffic inspection tool that can decode and display message traffic with minimal user effort and no impact on overall system performance. This and other features emphasize LCM's focus on simplifying both the development and debugging of message passing systems. In this paper, we explain the design of LCM, evaluate its performance, and describe its application to a number of autonomous land, underwater, and aerial robots.

I. 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 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.

II. Related Work

Interprocess communication is an extensively studied topic broad applicability. We direct our attention specifically towards its in the development of robotic systems, where the idea of dividing large systems into modules has become commonplace [3], [4].

There are several recurring themes in existing systems. Publish/subscribe models are the most commonly used [5], [3], [6], 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. A separate technical report describes a number of commonly used systems in greater detail [7].

Huang is at the Computer Science and Artificial Intelligence Laboratory at MIT, Cambridge, MA, USA. Olson is in the Computer Science and Engineering department at the University of Michigan, Ann Arbor, MI, USA. Moore is an IEEE member.

Service models provide a familiar programming model [8], [9], 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.

Existing systems are widely varied in terms of provided support for data marshalling. Some systems do not provide automatic marshalling tools, instead allowing free-form human-readable messages [5], or specifying a set of standardized messages and binary formats [10]. Several systems use an XDR-based marshalling system [11], [8], [3], though some implementations provide only partially automatic code generation. Language and platform support is typically limited, and with some systems [3], the users must manually keep a formatting string in sync with the message 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 [3], [4], [5]. 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.

III. Approach

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

A. 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 III-B 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.

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. 1 shows an example of two LCM message type definitions, and excerpts from the C bindings for these types are given in Fig. 2.

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

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

Fig. 1. 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.

```
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. 2. Excerpts from automatically generated C-language bindings for the types defined in Fig. 1. LCM also supports message type bindings for Python, Java, and MATLAB.

The LCM type specification was strongly influenced by the External Data Representation (XDR) [11], which is used by Sun Remote Procedure Calls (Sun/RPC) and perhaps most notably, the Network File System (NFS) [12]. Some XDR features are not supported by LCM due to the fact that they are rarely used, have portability issues, or invite user error, such as optional data (e.g., support for pointer chasing) and unions. LCM retains XDR's minimal computational requirements, allowing automatically-generated code to run on resource-constrained devices like microcontrollers.

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.

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.

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 (Fig. 1) is similar to that of a C struct.

The supported primitive data types are: `byte`, `int8_t`, `int16_t`, `int32_t`, `int64_t`, `float`, `double`, `string`, and `boolean`. LCM has adopted the C99 naming convention for integer types, which explicitly names the size of the data type in bits [13]. Making these sizes explicit (rather than defining an integer in terms of the host's word size, for example), is necessary for cross-platform compatibility, and encourages a deliberate choice of data range.

Strings 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, 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.

A member variable of an LCM type can also be another LCM type. Appropriate language-specific idioms are used to embed data from the second type in the first when the type is compiled to a language binding. For example, in C, the member is a pointer; in Java and Python, an object reference is used.

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 does not specify how the length of a variable-length array should be represented, with the result that different and incompatible methods have arisen across implementations. In comparison, LCM's method is both intuitive and flexible.

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`.

B. 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 guarantees 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.

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.

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` (Fig. 1) 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.

C. Communications

The communications aspect of LCM can be summarized as a publish-subscribe 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.

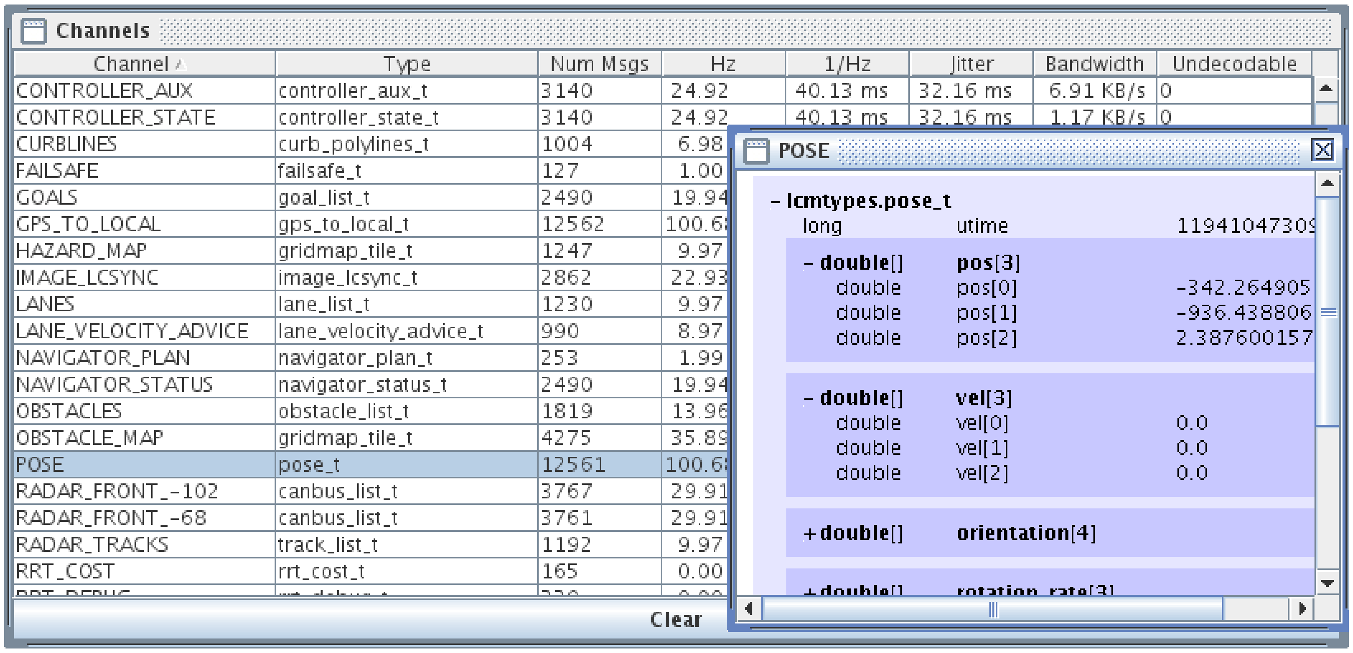
1) UDP Multicast: In typical publish-subscribe systems, a mediator process is used to maintain a central registry of all publishers, channels, and subscribers. Messages are then either routed through the mediator directly, or the mediator is used to broker point-to-point connections between a publisher and each of its subscribers. In both cases, the number of times a message is actually transmitted scales linearly with the number of subscribers. When a message has multiple subscribers, this overhead can become substantial.

The approach taken by LCM, in contrast, is simply to broadcast all messages to all clients. A client discards those messages to which it is not subscribed. Communication networks such as Ethernet and the 802.11 wireless standards make this an efficient operation, where transmitted packets are received by all devices regardless of destination.

UDP multicast provides a standardized way to leverage this feature, and its 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. Consequently, it does not require a centralized hub for either relaying messages or for "match making". A maximum LCM message size of 4 GB is achieved via a simple fragmentation and reassembly protocol.

The time-to-live parameter of multicast packets is used to control the scope of a network, and is most commonly set to 0 (all software modules hosted on the same computational device) or 1 (modules spread across devices on the same physical network).

2) Delivery Semantics: LCM provides a best-effort packet delivery mechanism and 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. This is similar to JAUS (which uses UDP), and TCP-based systems such as MOOS and Player. MOOS and Player both use a reliable transport (TCP), but allow messages to be dropped in order to manage situations in which the subscriber cannot keep up with a publisher.



Channel	Type	Num Msgs	Hz	1/Hz	Jitter	Bandwidth	Undecodable
CONTROLLER_AUX	controller_aux_t	3140	24.92	40.13 ms	32.16 ms	6.91 KB/s	0
CONTROLLER_STATE	controller_state_t	3140	24.92	40.13 ms	32.16 ms	1.17 KB/s	0
CURBLINES	curb_polylines_t	1004	6.98				
FAILSAFE	failsafe_t	127	1.00				
GOALS	goal_list_t	2490	19.94				
GPS_TO_LOCAL	gps_to_local_t	12562	100.6				
HAZARD_MAP	gridmap_tile_t	1247	9.97				
IMAGE_LCSYNC	image_lcsync_t	2862	22.93				
LANES	lane_list_t	1230	9.97				
LANE_VELOCITY_ADVICE	lane_velocity_advice_t	990	8.97				
NAVIGATOR_PLAN	navigator_plan_t	253	1.99				
NAVIGATOR_STATUS	navigator_status_t	2490	19.94				
OBSTACLES	obstacle_list_t	1819	13.96				
OBSTACLE_MAP	gridmap_tile_t	4275	35.89				
POSE	pose_t	12561	100.6				
RADAR_FRONT_-102	canbus_list_t	3767	29.91				
RADAR_FRONT_-68	canbus_list_t	3761	29.91				
RADAR_TRACKS	track_list_t	1192	9.97				
RRT_COST	rrt_cost_t	165	0.00				

```
POSE

- lcmtypes.pose_t
    long          utime           11941047309
  - double[]      pos[3]
      double      pos[0]          -342.264905
      double      pos[1]          -936.438806
      double      pos[2]          2.387600157
  - double[]      vel[3]
      double      vel[0]          0.0
      double      vel[1]          0.0
      double      vel[2]          0.0
  + double[]      orientation[4]
```

Fig. 3. 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.

In general, a system that has significant real-time constraints, such as a robot, may in many cases prefer that a lost packet (e.g., a wheel encoder reading) simply be dropped so that it does not delay future messages. LCM reflects this in its default mode of operation; higher level semantics may still be implemented on top of the LCM message passing service.

D. *Tools*

To assist development of modular software systems, LCM provides several tools useful for logging, replaying, and inspecting traffic. Together, they allow a developer to rapidly and efficiently analyze the behavior and performance of an LCM system.

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. We note that the logging and playback programs are not attributed special status in LCM; one simply subscribes to all channels, and the other publishes messages from a data file.

1) 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[1] 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. 3 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

[1]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.

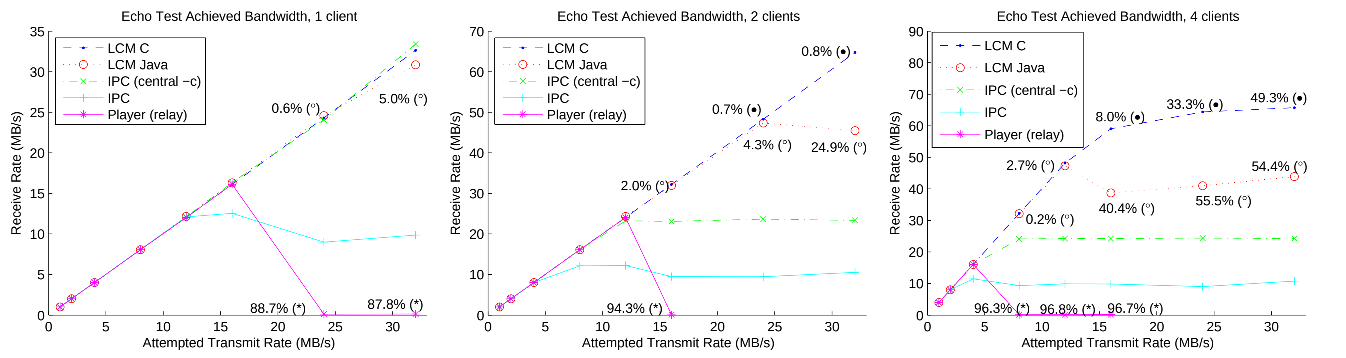


Fig. 4. Bandwidth results from an echo test with 1, 2, and 4 clients. Each client echoes messages transmitted by a single sender, and the bandwidth of successful echoes as detected by the original sender are shown. Message loss rates (messages with no received echo) are shown in parentheses when nonzero.

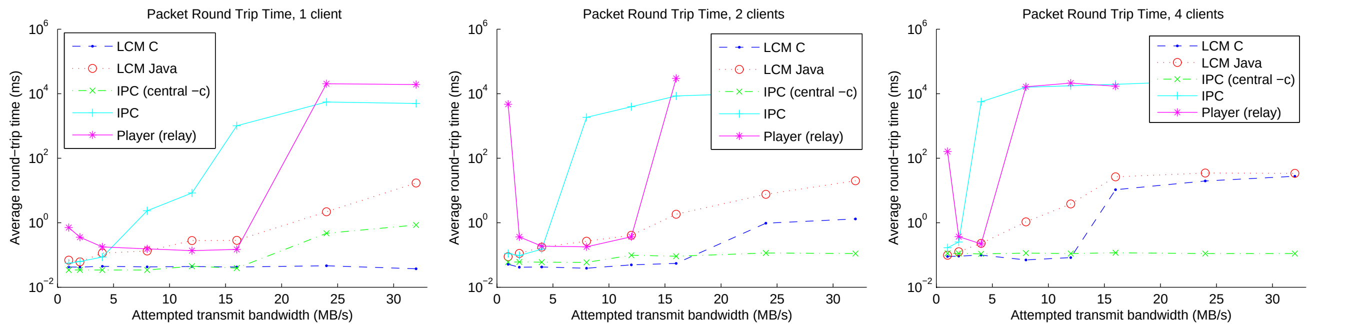


Fig. 5. Mean message round-trip times for the echo test in Fig. 4 with 1, 2, and 4 clients. Average round-trip times are shown on a log scale; these times do not reflect lost packets (which essentially have an infinite round-trip time). Both LCM implementations offer low latency. While IPC (using central -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. 4).

passively observes and analyzes traffic, lcm-spy can provide this insight with absolutely no impact on the system performance.

IV. Performance

One way to measure the interprocess communication performance of LCM is by examining its bandwidth, latency, and message loss rate under various conditions. Figs. 4 and 5 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. Figs. 4 and 5 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, 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 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, 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 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.

A. *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[2] 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. The LCM type definition used is:

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

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 6. 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.

V. Case Studies

Since its development, LCM has been used as the primary communications system for a number of robotics research

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

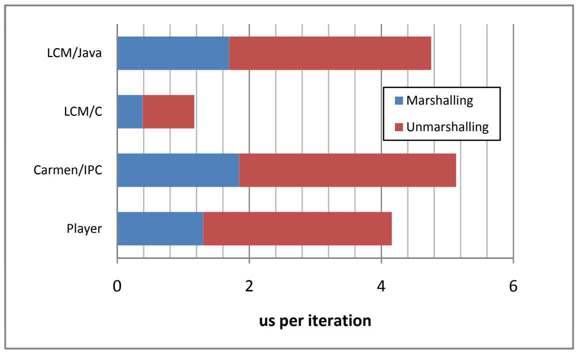


Fig. 6. Marshalling performance. Performance (in microseconds per iteration) is shown for four different marshalling implementations. The LCM implementation in C is more than four times faster than the next fastest marshaller. Notably, the LCM Java implementation, despite being written in pure Java, is comparable to the Carmen and Player implementations.

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 I

LCM traffic summary on MIT Urban Challenge vehicle

platforms operating in real environments. Here, we describe some examples of how LCM was used to assist the development of a robotics research system.

A. *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 suburban 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 [14].

At any given point in time on the MIT vehicle, 70 separate modules were in active operation, spread across 10 networked workstations. 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 I provides a detailed breakdown of messaging rates and the bandwidth used by various modules on the vehicle.

Messages ranged from very small updates to camera images and obstacle maps up to several megabytes in size.

Some, 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.

B. Land, Underwater, and Aerial Robots

Since the Urban Challenge, LCM has been applied to a number of other robot research platforms such as small indoor wheeled robots, arm manipulators, quadrotor helicopters [15], and an autonomous forklift [16]. In many cases, modules used in one system were easily transitioned to others by ensuring that the LCM messages they needed for correct operation were present on the target robot.

In one underwater multi-vehicle research project [17], each vehicle contains on-board sensors, thrusters, and a computer for data processing and vehicle control. Despite a significantly different application domain from the Urban Challenge, 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.

VI. 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.

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`. It is supported on Microsoft Windows XP/Vista/7 and all POSIX.1-2001 compliant platforms (GNU/Linux, OS/X, FreeBSD, etc.)

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.

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

[3] M. Montemerlo, N. Roy, and S. Thrun, "Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (carmen) toolkit," in *Proc. Int. Conf. on Intelligent Robots and Systems*, vol. 3, Las Vegas, NV, USA, October 2003, pp. 2436–2441.

[4] Willow Garage, "Robot Operating System," http://www.ros.org.

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

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

[7] A. S. Huang, E. Olson, and D. Moore, "Lightweight communications and marshalling for low latency interprocess communication," Massachusetts Institute of Technology, Tech. Rep. MIT-CSAIL-TR-2009-041, 2009. [Online]. Available: http://hdl.handle.net/1721.1/46708

[8] T. H. Collet, B. A. MacDonald, and B. P. Gerkey, "Player 2.0: Toward a practical robot programming framework," in *Proc. Australasian Conf. on Robotics and Automation*, Sydney, Australia, Dec. 2005.

[9] 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

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

[11] 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

[12] 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

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

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

[15] 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

[16] S. Teller *et al.*, "A voice-commandable robotic forklift working alongside humans in minimally-prepared outdoor environments," in *Proc. Int. Conf. Robotics and Automation*, Anchorage, AL, USA, May 2010.

[17] 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.