MCT User Guide

$Rev: 3$

This user guide gives a conceptual overview of the features MCT provides, and an architectural overview of how its components fit together. Application design is not covered, and API use is not duplicated here; that is documented by the `libmct` API manual.

Contents

Bubblephone Limited
Sussex Innovation Centre
Science Park Square
University of Sussex
Brighton BN1 9SB
Tel +44 (0) 1273 704 535
http://www.bubblephone.com/

Bridging

Bridges allow the flow of network traffic between two endpoints. Each bridge is identified by a unique session ID, allocated on creation of the bridge. Session IDs are re-used when bridges are destroyed, and are unique for the lifetime of the bridge.

A bridge has a source and destination endpoint, termed `src` and `dst` respectively. For some protocols traffic may flow bidirectionally; this is not true in the general case. Endpoints are at OSI network layer 4, which for IP corresponds to TCP and UDP ports.

The `src` and `dst` endpoints need not exist in the same address or protocol families; MCT may act as a gateway between different transport protocols. However both `src` and `dst` must have the same socket type (e.g. `SOCK_STREAM` or `SOCK_DGRAM`).

Connectionless protocols

Several protocols are connectionless, and transfer atomic messages without maintaining state between peers. For IP systems, `SOCK_DGRAM` messages are implemented by UDP. However MCT is not IP-specific, and so other connectionless protocols are also supported, notably including `SOCK_RDM` (Reliable datagrams).

Connectionless messages are unidirectional. If two-way communication is required, two bridges will need to be instantiated.

`AF_INET` (IPv4) is used for the examples here. The IP and port numbering is representative. The MCT daemon is both protocol and address family agnostic, although the `libmct` API only provides a subset of those protocols. These are noted specifically in the “Caveats” sections of the relevant API manpages. In particular, `AF_INET6` (IPv6) is also provided.

Addresses map to network interfaces per the OS configuration; this is the responsibility of the system administrator, and MCT does not attempt to interfere with address allocation.

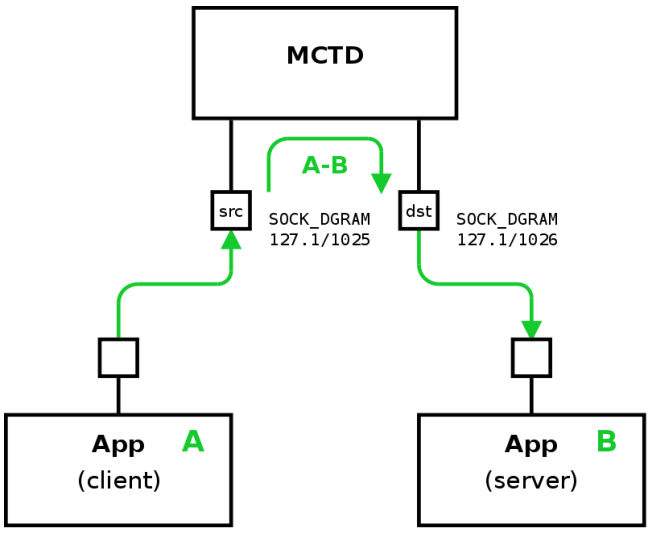

Figure 1. Client-to-server bridge

Figure 1 illustrates two representative applications with a single bridge from UDP port `1025` to UDP port `1026` on the same IP, `127.0.0.1`.

These applications are not part of MCT; they may be user-facing, or themselves daemons, or something else as suits the particular requirements of the system.

Note that because the bridge is connectionless, any peers with a route may send datagrams to the `src` endpoint `12.7/1025`, as for `SOCK_DGRAM` in general,

Figures 2 and 3 illustrate multiple peers, bridging connectionless messages from multiple clients to one server and one client to multiple servers, respectively.

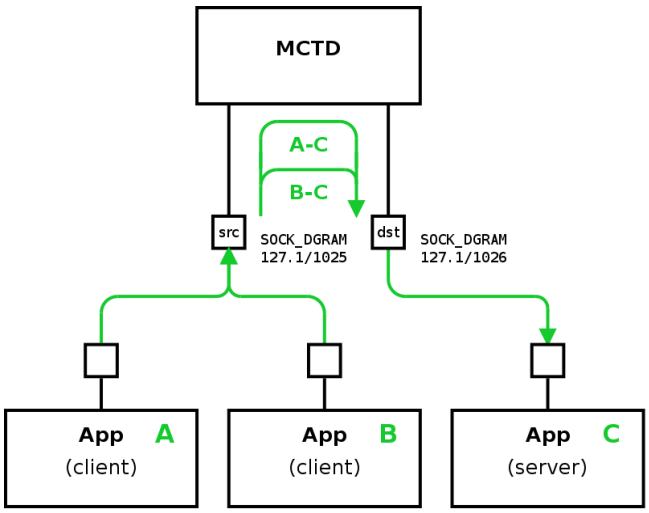

Figure 2. Multiple clients to one server

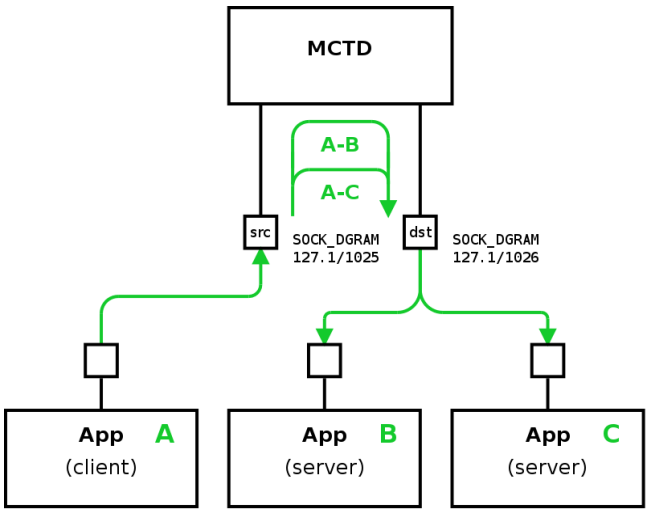

Figure 3. One client to multiple servers

This operation makes use of the `SOCK_DGRAM` semantics permitting message delivery without a requirement for confirmation back to the originator. In the case of two client applications, there are two originating sources of messages, and these are both unaware of each other. Likewise the receiving server is not aware that its incoming traffic is from two separate sources; messages are delivered with the `dst` endpoint marked as the peer address, retrievable by the POSIX `getpeername()` interface.

Although this arrangement for connectionless protocols is provided by the MCTD daemon, it is not currently exposed through the `libmct` interface, which requires `src-dst` pairs to be unique.

Connection Orientated Protocols

Stateful connections provide implementations for several reliable, ordered protocols (Notably SOCK_STREAM and SOCK_SEQPACKET). For IP, SOCK_STREAM is implemented by TCP.

As for listening sockets in SOCK_STREAM connections, the bridge itself isn't used for the streams' traffic flow. Rather, it exists to provide a destination for incoming connections to be accepted.

A new session is created when a peer connects. Multiple peers may connect to the same `src` endpoint, and a new session ID is allocated for each. These sessions are the connections over which traffic is transferred.

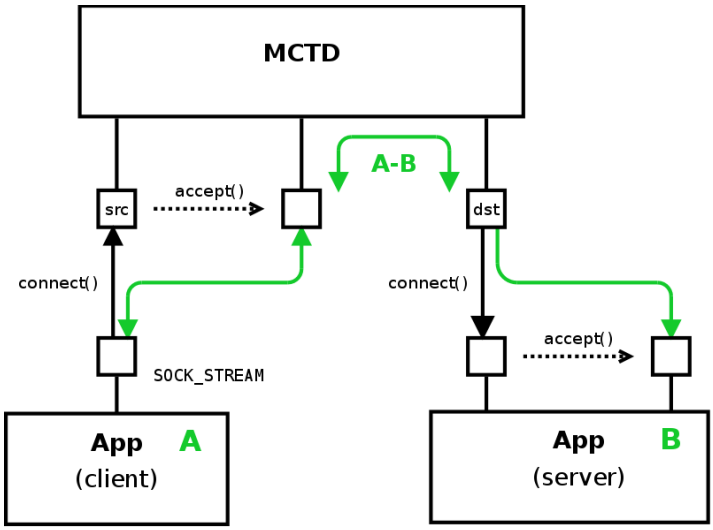

Figure 4. Accepting peers

Events illustrated by Figure 4 occur in the following sequence:

1. Application A connects to the `src` endpoint;
2. MCTD calls `accept()` for this connection, and in turn calls `connect()` to application B;
3. Application B accepts from the `dst` endpoint;
4. MCTD allocates a session ID for the A-B connection;
5. Traffic flows between A and B's newly-accepted peer.

Traffic for connection-oriented protocols is bidirectional.

Multicast

Both endpoints of a bridge may be independently joined to zero or more multicast groups, and one endpoint may be joined to multiple groups.

Furthermore, both endpoints may be joined to the same group; this is usually not desirable. In general, the multicast implementation of underlying operating systems does little to prevent creating routing loops and excessive packet duplication. The one mechanism provided is exposed as an interface to set packet TTL, which must be configured per application requirements, as the default is to drop all traffic for a group. This reflects the behaviour of the underlying network stack.

Figures 5 and 6 illustrate group membership of the `src` and `dst` endpoints respectively.

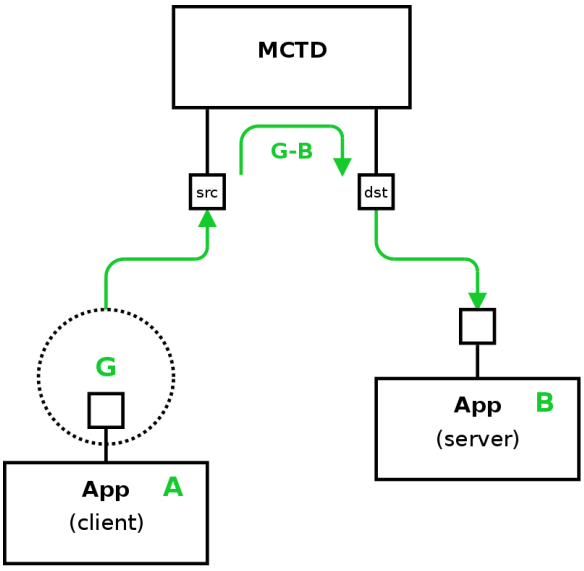

Figure 5. `src` Group membership

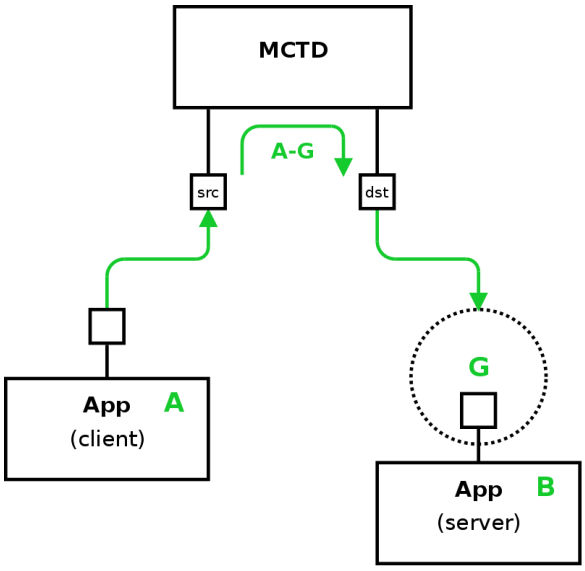

Figure 6. `dst` Group membership

For the G-B bridge, any node joined to group G may multicast to the entire group; there is no restriction for traffic to originate solely from Application A. Likewise for the A-B bridge, any node joined to group G may receive traffic from Application A, not just Application B

IPC

Applications which link against the `libmct` run as separate processes to the MCT daemon. The library provides control over MCTD through Inter-Process Communication; the IPC protocol is private, and is not intended to be implemented by anything other than `libmct` and MCTD.

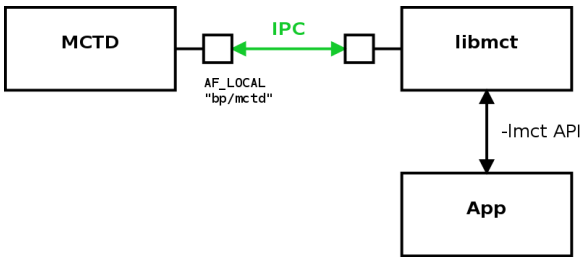

Figure 7. IPC

IPC between `libmct` and MCTD occurs over an `AF_LOCAL` socket, illustrated by Figure 7. This socket is named by both the daemon command line and the `mct_init()` interface for `libmct`; multiple daemons (not illustrated) may co-exist independently by selecting unique names for their IPC sockets, and may be connected to either by independent applications or by calling `mct_init()` multiple times from within the same application.

The IPC protocol is versioned, in order to identify accidentally linking against a version of `libmct` which is not supported by the running daemon or vice versa.

The protocol is message based, and intended to be extensible with new message types as required for future feature additions. Message delivery is reliable, and successful operation is confirmed before `libmct` API functions return to the caller. Errors due to IPC transport are presented through the `libmct` API, documented by the `mct_ipc(3)` manpage.

Asynchronous Events

The libmct API provides notification of events which occur asynchronously relative to the MCT daemon operation. These provide a mechanism for applications to be informed of various situations due to activity outside of an application's control, such as connecting peers.

Events are categorised by type. The exact set of event types provided is intended to be extended as per application requirements.

Events are numbered sequentially, such that an application can identify if it has missed an event, for example by disconnecting its MCT handle and reconnecting it. Sequence numbers roll over, rather like SNMP counters.

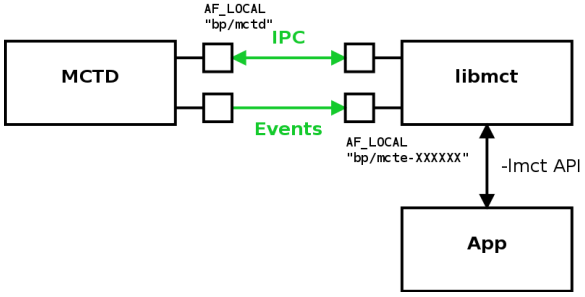

Figure 8. IPC with events

Multiple instances of libmct may connect to the same daemon concurrently. As with multiple daemons, instances may be either separate applications (illustrated) or within the same application.

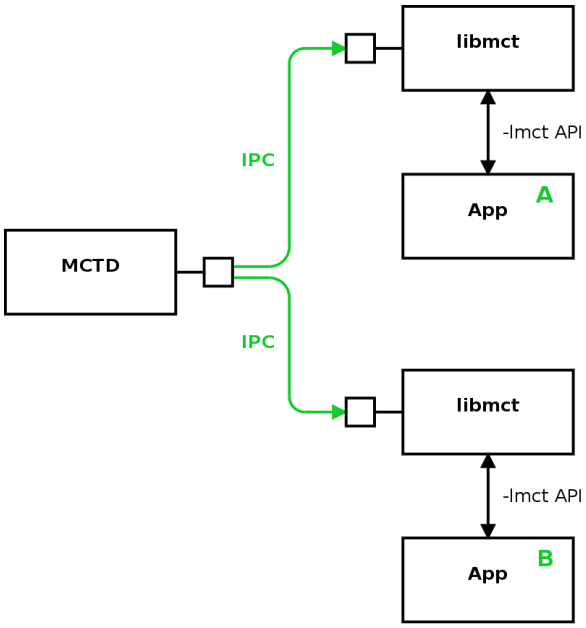

Figure 9. IPC with multiple instances

The libmct implementation and IPC communication are thread safe, but do not provide locking. It is the caller's responsibility to ensure two calls to the same MCT handle (as returned from `mct_init`) are not undertaken simultaneously. However calls to separate handles may occur simultaneously if called in separate threads, just as they may from separate processes.

Appendix A. FFI API Wrapping

The `libmct` API requires only standard constructs, defined by either ISO C headers or by POSIX. These are intended to be straightforward to represent natively in other languages, as largely these constructs (such as mappings for types) already exist.

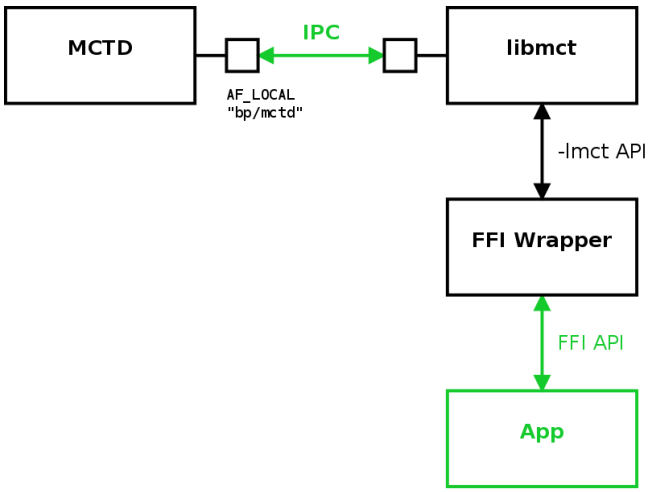

Figure 10. FFI

Here the suggested approach is for an Foreign Function Interface to present the `libmct` API faithfully – which an application may use directly (as highlighted in Figure 10) – or to then wrap that representation with a model which better suits the native language (not illustrated).

Foreign languages can present the API more naturally, e.g. mapping MCT concepts to constructs such as classes, message passing, and exceptions. The advantage of a two-stage approach is that the FFI wrapper may be encapsulated, and reduce the overall C footprint and the majority of wrapping code may be in the interface's native language.

No FFI wrappings are provided as part of the MCT package.

Appendix B. Security Considerations

The MCT daemon provides several features which do not affect functionality, but instead serve to isolate operational code with security in mind.

Firstly the separation of `libmct` from the daemon itself (which is responsible for maintaining working connections); the only interaction from applications is over the IPC protocol. Instances of the `libmct` library reside within the application's address space, and so if `libmct` is compromised then the daemon remains unaffected; control over the daemon still requires use of IPC.

The daemon is able to run attached to a console (for debugging), but for normal operation should be permitted to daemonise itself. This is the usual UNIX approach of forking twice, becoming inherited by `init`, and closing its file descriptors.

The process of daemonisation may also include dropping privileges to run as a user which does not have permission to do anything other than accept and create sockets. This should be used in conjunction with the `chroot` feature, which removes the ability for the running process to access the filesystem during normal operation. The chrooted filesystem should not be made writeable to the unprivileged user. These features are documented in the `mctd(1)` manpage.

The daemon does not call the system's name resolver. In part because on some systems this requires access to `/dev` which would require populating the chrooted filesystem. But primarily because this is a feature which is offloaded not only to the `libmct`-side of IPC but to the application itself. Resolved addresses are passed in by the standard POSIX form of `sockaddr` structures. This serves to isolate use of external systems from the operation of the daemon.

Appendix C. Design Rationale

Requisites

- Auditable codebase suitable for use in defence environments.
- Automated control via a programmatic interface.
- Target POSIX systems. These may be embedded and hence relatively low on resources.
- As closed to fixed-size resource footprint as possible. In particular, memory consumption is not to increase to unwieldy sizes proportionally to the number of connections handled. Likewise use of the filesystem is not to accumulate logs.
- Low latency for traffic, and low throughput overhead.
- The driving use-case is an intention for all connections on a physical machine to go via this product. MCT in general does not require this; it cooperates with existing systems, rather than enforce their integration.

Implementation Choices

- Userspace implementation for straightforward portability.
- Therefore opted for daemon for privilege separation.
- Daemon provides arbitrary bridges; this generalises the requirements.
- No tunnelling; throughput is unaffected.
- Control over private extensible IPC protocol.
- library provides interface to IPC; multiple instances may connect simultaneously
- Library lives in application space, and may be short-lived (e.g. for CLI commands) or long-lived (e.g. if the application is itself a daemon).
- Asynchronous Event notification.
- Integrate with standard OS configuration (IPs map to interfaces, lsof, etc).
- Logging is provided to the standard Syslog facility.

Commentary on API Design

- Grouped in a roughly OOish manner.
- Exposed as ISO C90 and POSIX.
- intended for one-to-one FFI wrapping, with full OO FFI-side.
- Convenience API separated as it's not part of MCT proper.
- IPC (and therefore the `libmct` API) only need expose require MCTD functionality.