# Auto-generated xdcomms system model and specification

Columbia University, as part of the GAPS-CLOSURE project

March 21, 2023

## Contents

# 1 Introduction

## 1.1 About xdcomms and the GAPS-CLOSURE project

==TODO==

## 1.2 About this document

==TODO==

# 2 Summary

## 2.1 High-level overview of xdcomms auto-generation

==TODO==

## 2.2 Goals of xdcomms design and implementation

==TODO==

# 3 Information model

In this section we formally model all of the datatypes and classes used in the xdcomms system, and provide intuition as to their purpose; the application control flow and function behavior is fully specified in Section 4.

## 3.1 Summary diagram

The following UML diagram provides a bird's eye summary of the information model. ==TODO==

## 3.2 Primitives and aliases

The following aliases are used in our information model.

==TODO== (Section 3.3).

```
MuxTag :: int
SecTag :: int
TypTag :: int
GTag :: (MuxTag, SecTag, TypTag)
```

An Enclave is a name for a set of devices operating in a closed network at the same security level. A Level is a name given to a security level for an application endpoint, which is used to specify what data the endpoint can receive from which other endpoints, and across which communication channels.

```
Enclave :: string
Level :: string
```

==TODO==

```
DataType :: string
Status :: int
RequestID :: int
ResponseID :: int
Timestamp :: int
```

==TODO== (Section 3.4).

```
AppEndpt :: URI
DevType :: string
FHdl :: int
TagDev :: (GTag, Device)
```

## 3.3 Cross-domain data format

This subsection describes the format of program data which can be transferred from one application endpoint to another via xdcomms, and the intermediate representations of said data.

### 3.3.1 The `Data` type

xdcomms uses a simple type, `Data`, to capture a collection of program data which is eligible to be transferred cross-domain to another application endpoint. It is isomorphic to a struct in C; the purpose is to *lift* valid, serializable structs into the xdcomms informaton model.

```
type Data =
    dtype    :: DataType  // string identifier for this data format.
    contents :: [CValue]  // ordered list of fields, where CValue refers to
                          // a typed value in C that is a scalar, vector,
                          // or serializable struct.
```

We restrict `CValue` to *serializable* datatypes, meaning a `CValue` is not a pointer, or, if it is a struct, the struct fields are all serializable (i.e. not pointers). Because application endpoints are physically and memory isolated, pointers cannot be transferred cross-domain. Fixed-length arrays, however, are valid `CValue`s (because fixed-length arrays are stored by value in C structs).

### 3.3.2 The `Marshalled` type

Serialization of data into a packet requires awareness of the network and the protocol between the communicating devices. There is, however, device-independent information that must be appended to the data being transferred; we refer to this process as *marshalling.*

The `Marshalled` type captures data which is ready for serialization; it contains the original data, along with a *trailer* that stores a sequence number and error-correction codes:

```
type Marshalled =
    mcontents :: Data
    trailer   :: Trailer

type Trailer =
    seq :: uint32_t
    rqr :: uint32_t
    oid :: uint32_t
    mid :: uint16_t
    crc :: uint16_t
```

Marshalled and Trailer are *packed* so that their fields have no padding, for ease of serialization. ==PERATON: What are the other trailer fields for?==

### 3.3.3 The `Serialized` type

Before being sent cross-domain, data converted into an in-memory `Marshalled` struct with a sequence number and error correction codes must additionally be:

- Tagged with a GTag.
- Given a unique ID to distinguish it from other incoming request or response packets at the destination.
- Serialized into a stream of bytes in network order.
- Coerced into the packet format expected at the destination device.

This transformation is mediated through the `Serialized` type:

```
type Serialized =
    tag   :: GTag
    bytes :: [uint8_t]
    reqid :: RequestID
    rspid :: ResponseID
```

The GAPS tag, of type `GTag`, is a tuple of three tags: a `MuxTag` identifying the source and destination applications, a `SecTag` identifying the security levels between which the data is being transferred, and a `TypTag` encoding the `DataType` as an integer (so that it can be unmarshalled at the destination). So for `GTag` we have the type `GTag : (MuxTag, SecTag, TypTag)`.

The request ID `reqid` is set if this communication is a request, and the response ID `rspid` is set if this communication is a reply to a request. ==PERATON: Where are the request and response ID set? How exactly are they used?==

Conversion of a `Marshalled` struct proceeds in several steps. First, the `Marshalled` field is copied into a parallel (packed) struct, in which every field (including those in the trailer) is represented as bytes in *network byte order*. This requires that floats be converted into a device-independent bytestring format. Then, this parallel struct is cast to a void pointer and treated as an array of bytes, stored in the bytes field of the `Serialized` struct. ==PERATON: which part of this is device-aware? The codec functions don't seem to perform any device-specific packetizing. Is this hiding in zmq send?==

Conversion between a `Marshalled` struct and its corresponding `Serialized` struct is mediated by the `serialize()` and `deserialize()` functions in the device-aware `Codec` class (Section 3.4).

## 3.4 The hardware interface

==TODO==

### 3.4.1 The `HALConfig` class

==TODO==

### 3.4.2 The `Device` class

==TODO==

### 3.4.3 The `Codec` class

==TODO==

## 3.5 The network abstraction

==TODO==

### 3.5.1 The `Binding` type

==TODO==

### 3.5.2 The `XDContext` type

==TODO==

### 3.5.3 The `Wrapper` class

==TODO==

### 3.5.4 The `Handler` class

==TODO==

## 3.6 The application model

==TODO==

### 3.6.1 The `MasterSequence` class

==TODO==

### 3.6.2 The `EventQueue` class

==TODO==

### 3.6.3 The `RPCTransaction` class

==TODO==

### 3.6.4 The `HAL` class

==TODO==

### 3.6.5 The `App` class

==TODO==

### 3.6.6 The `AppThread` class

==TODO==

# 4 Behavioral specification

==TODO==

## 4.1 Control flow

==TODO (diagram)==

## 4.2 Function-level contracts

==TODO==

### 4.2.1 `Wrapper.marshall()`

The `marshall()` function is called by the `Wrapper` to marshall the incoming arguments (the arguments to the cross-domain function in the original, unpartitioned program) into one serializable data structure with error correction and a sequence number.

Arguments: `d :: Data`, `req_counter: int` (d is unpacked and may represent multiple or no arguments)

Return value: `m :: Marshalled`

Pre-conditions:

- The value of req_counter is either non-negative or `INT_MIN`.
- The `Marshalled` struct defines, in the same order as the arguments to `marshall(}`, one field of the same type and name as each argument, followed by a `Trailer`.

Post-conditions:

- For each incoming argument, the corresponding field in `m` is set to the value of that argument.
- The trailer of `m` has its `seq` field set to the value of `req_counter`.

# 5 Generator operation

==TODO==

## 5.1 Inputs: The GEDL

==TODO==

## 5.2 Outputs

## 5.3 Whole-system correctness properties and proofs

==TODO==

# 6 References