## Overview

pvAccess is a high-performance network communication protocol for designed signal monitoring and also for high-level applications. It is a successor of of EPICS Channel Access.

TCP/IP is used for data transmission, UDP/IP for discovery (discovery over TCP/IP can be easily implemented).

pvAccess heavily depends on pvData – all the data is sent as pvData objects. Protocol is optimized to send minimum amount of data. It supports segmented messages and allows sending huge amount of data using small buffers (there is no limitation on maximum message size limited by send/receive buffer).

The pvAccess protocol definition consists of three major parts:

* a set of data encoding rules that determine how the various data types are de-/serialized
* a number of message types that are interchanged between client and server, together with rules as to what message is to be sent under what circumstances
* a set of rules that determine how client and server agree on a particular protocol and encoding version

## Data Encoding

The key goals of the pvData data encoding are simplicity and efficiency. In keeping with these principles, the encoding does not align primitive types on word boundaries and therefore eliminates the wasted space and additional complexity that alignment requires. The pvAccess data encoding simply produces a stream of contiguous bytes; data contains no padding bytes and need not be aligned on word boundaries.

Data is currently encoded using big-endian byte order (inheritance of initial implementation in Java), however server’s endianess will be used in future. In order to allow all the intermediates to forward the data without requiring it to be unmarshaled (the intermediates can forward requests by simply copying blocks of binary data), each message should indicate endiannes.

### Sizes

Many of the types involved in the data encoding, as well as several protocol message components, have an associated size or count. A size is a non-negative number. Sizes and counts are encoded in one of two ways:

1. If the number of elements is less than 254, the size is encoded as a single byte indicating the number of elements.
2. If the number of elements is greater than or equal to 254, the size is encoded as a byte with value 254, followed by an 32-bit integer indicating the number of elements.
3. “null” arrays/strings are encoded as byte with value 255.

Using this encoding to indicate sizes is significantly cheaper than always using an 32-but integer to store the size, especially when marshaling sequences of short strings: counts of up to 253 require only a single byte instead of four. This comes at the expense of counts greater than 253, which require five bytes instead of four. However, for sequences or strings of length greater than 253, the extra byte is insignificant.

### Basic Types

The basic types are encoded as shown in. Integer types (byte, short, int, long) are represented as two’s complement numbers, and floating point types (float, double) use the IEEE standard formats.

|  |  |
| --- | --- |
| Type | Encoding |
| boolean | A single byte with value non-zero value for true, 0 for false |
| byte | Signed 8-bit integer. |
| short | Signed 16-bit integer. |
| int | Signed 32-bit integer. |
| long | Signed 64-bit integer. |
| float | 32-bit float (23-bit fractional mantissa, 8-bit exponent, sign bit) |
| double | 64-bit float (52-bit fractional mantissa, 11-bit exponent, sign bit) |

Encoding for basic types.

### Strings

Strings are encoded as a size, followed by the string contents in UTF-8 format array of bytes. Strings are not NULL-terminated. An empty string is encoded with a size of zero.

### Arrays

Arrays are encoded as a size representing the number of elements in the array, followed by the elements encoded as specified for their type.

### BitSets

BitSet is a pvData internal data type that represents bits. It is serialized as a byte array.

### Introspection data

Each pvData data instance (i.e. PVField) has introspection description (i.e. Field). PVField is encoded as raw values (no other overhead), Field encodes information of what data-type(s) PVField is. Since many different PVField instances share same introspection description it can be cached to avoid sending it over-and-over over the wire. Thus, Field encoding starts with a byte describing the following encoding: it can be NULL, ID only and ID with actual Field serialization.

ID, encoded as short, is a key used to cache Field descriptions.

|  |  |  |
| --- | --- | --- |
| Field Encoding | Type | Description |
| 0xFF | **NULL\_TYPE\_CODE** | NULL. |
| 0xFE + ID | **ONLY\_ID\_TYPE\_CODE** | Serialization contains only an ID (that was assigned by one of the previous **FULL\_WITH\_ID** descriptions). |
| 0xFD + ID + FieldDesc | **FULL\_WITH\_ID\_TYPE\_CODE** | Serialization contains an ID (that can be used later, if cached) and full interface description. |

Field encoding.

Actual Field introspection description (FieldDesc) is encoded as a byte that consists of 2 nibbles (4-bits). Upper nibble (MSBs) defines Field type (i.e. Type), low nibble (LSBs) defines data type (i.e. ScalarType) for scalar and scalarArray types. The byte is then followed by Field instance name (encoded as string). Moreover, structure and structureArray types require more data: structure requires array of Field-s (size followed by Field-s as described above), structureArray is similar just that it in addition requires name of the structure.

|  |  |  |
| --- | --- | --- |
| FieldDesc Encoding | Type | Description |
| 0x00 | ScalarType (0b0000xxxx) +  name | **scalar** | Scalar. |
| 0x10 | ScalarType (0b0001xxxx) +  name | **scalarArray** | Array of scalars. |
| 0x20 (0b00100000) +  name + Field[] | **structure** | Structure. |
| 0x30 (0b00110000) +  name + (structure name + Field[]) | **structureArray** | Array of structures. |

FieldDesc encoding.

|  |  |
| --- | --- |
| Encoding | ScalarType |
| 0 (0b0000) | **boolean** |
| 1 (0b0001) | **byte** |
| 2 (0b0010) | **short** |
| 3 (0b0011) | **int** |
| 4 (0b0100) | **long** |
| 5 (0b0101) | **float** |
| 6 (0b0110) | **double** |
| 7 (0b0111) | **string** |

ScalarType nibble encoding.

#### Example

Introspection description of the following structure

structure test1

structure timeStamp

long secondsPastEpoch

int nanoSeconds

structure[] value

structure org.epics.ioc.test.testStructure

double value

structure location

double x

double y

string factoryRPC

structure arguments

int size

structure element

double value

structure alarm

int severity

string message

structure timeStamp

long secondsPastEpoch

int nanoSeconds

is encoded as

FD 00 01 20 05 74 65 73 74 31 05 FD 00 02 20 09 .... .tes t1.. ....

74 69 6D 65 53 74 61 6D 70 02 04 10 73 65 63 6F time Stam p... seco

6E 64 73 50 61 73 74 45 70 6F 63 68 03 0B 6E 61 ndsP astE poch ..na

6E 6F 53 65 63 6F 6E 64 73 FD 00 03 30 05 76 61 noSe cond s... 0.va

6C 75 65 20 6F 72 67 2E 65 70 69 63 73 2E 69 6F lue org. epic s.io

63 2E 74 65 73 74 2E 74 65 73 74 53 74 72 75 63 c.te st.t estS truc

74 75 72 65 02 06 05 76 61 6C 75 65 FD 00 04 20 ture ...v alue ...

08 6C 6F 63 61 74 69 6F 6E 02 06 01 78 06 01 79 .loc atio n... x..y

07 0A 66 61 63 74 6F 72 79 52 50 43 FD 00 05 20 ..fa ctor yRPC ...

09 61 72 67 75 6D 65 6E 74 73 01 03 04 73 69 7A .arg umen ts.. .siz

65 FD 00 06 20 07 65 6C 65 6D 65 6E 74 03 06 05 e... .el emen t...

76 61 6C 75 65 FD 00 07 20 05 61 6C 61 72 6D 02 valu e... .al arm.

03 08 73 65 76 65 72 69 74 79 07 07 6D 65 73 73 ..se veri ty.. mess

61 67 65 FE 00 02 age. ..

### Data

Data (i.e. PVField) basically encodes only its data. The only exception is structureArray. Before each structure of an array there is a boolean flag indicating whether structure is null or not.

Each structure can have a BitSet instance defining what subset of all structure fields have been actually serialized. This allows partial serialization of structures, e.g. sending only fields that have changed and not entire structure. Each node of a structure corresponds to one bit; if a bit is set its corresponding field has been serialized, otherwise has been ignored. BitSet does not apply to array elements.

#### Example

|  |  |
| --- | --- |
| bit# | field |
| 0 | structure |
| 1 | structure timeStamp |
| 2 | long secondsPastEpoch 1296564296 |
| 3 | int nanoSeconds 819000000 |
| 4 | structure[] value |
|  | structure org.epics.ioc.test.testStructure |
|  | double value 100 |
|  | structure location |
|  | double x 0 |
|  | double y 0 |
|  | structure org.epics.ioc.test.testStructure |
|  | double value 200 |
|  | structure location |
|  | double x 5 |
|  | double y 10 |
| 5 | string factoryRPC org.epics.ioc.support.rpc.ExampleChannelRPCFactory |
| 6 | structure arguments |
| 7 | int size 2 |
| 8 | structure element |
| 9 | double value 0 |
| 10 | structure alarm |
| 11 | int severity 0 |
| 12 | string message |
| 13 | structure timeStamp |
| 14 | long secondsPastEpoch 0 |
| 15 | int nanoSeconds 0 |

The structure above requires 16 bits, i.e. 2 bytes. Remark: for performance reasons BitSet implementation stores bits as longs, however serialization is optimized to send only least possible number of bytes.

If bit on a structure node is set then all its fields has to be de-/serialized.

The example below is a simple example where entire structure is sent, i.e. bit 0 is set. NOTE: first six bytes are erased since they are not part of structure serialization.

01 01 00 00 00 00 4D 48 00 48 .. .... MH.H

30 D0 F2 C0 02 01 40 59 00 00 00 00 00 00 00 00 0... ..@Y .... ....

00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 40 .... .... .... ...@

69 00 00 00 00 00 00 40 14 00 00 00 00 00 00 40 i... ...@ .... ...@

24 00 00 00 00 00 00 32 6F 72 67 2E 65 70 69 63 $... ...2 org. epic

73 2E 69 6F 63 2E 73 75 70 70 6F 72 74 2E 72 70 s.io c.su ppor t.rp

63 2E 45 78 61 6D 70 6C 65 43 68 61 6E 6E 65 6C c.Ex ampl eCha nnel

52 50 43 46 61 63 74 6F 72 79 00 00 00 02 00 00 RPCF acto ry.. ....

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .... .... .... ....

00 00 00 00 00 00 00 .... ...

## Protocol Messages

The pvAccess protocol uses two protocol message types:

* control messages (flow control, no payload for now) and
* data messages (requests and their responses).

As with the data encoding described, protocol messages have no alignment restrictions. Each message consists of a message header and (optionally) a message payload that immediately follows the header.

### Message header

Each protocol message has a fixed 8-byte header that is encoded as if it were the following structure:

struct pvAccessHeader {

byte magic;

byte version;

byte flags;

byte messageType;

int payloadSize;

};

|  |  |
| --- | --- |
| Member | Description |
| magic | pvAccess protocol magic, 0xCA. |
| version | Protocol version, 2 nibbles (major, minor). |
| flags | Message flags. |
| messageType | Message type (i.e. create, get, put, process, etc.). |
| payloadSize | Message payload size (in bytes). |

pvAccess header members.

|  |  |
| --- | --- |
| bit | Description |
| 0 | Data/Control [0/1] message. |
| 1,2,3 | Unused. |
| 4 | First message (of set of segmented messages). |
| 5 | Last message (of set of segmented messages). |
| 6,7 | Unused. |

pvAccess header flags description.

NOTE: If bits 4 and 5 are both sets this indicates message between first and last message of segmented messages. If none set then message is not segmented.

### pvAccess requests

#### Beacon (0x00)

Send over UDP to clients. Beacons are only used to detect new servers and server restarts.

struct beaconMessage {

short beaconSequenceId;

long startupTimeSeconds;

int startupTimeNanos;

byte[16] serverAddressIPv6;

short serverPort;

FieldDesc serverStatusIF;

[if serverStatusIF != null] PVField serverStatus;

};

|  |  |
| --- | --- |
| Member | Description |
| beaconSequenceId | Beacon sequence ID (counter w/ rollover). Can be used to detect UDP routing problems. |
| startupTimeSeconds | Server startup time (seconds past 1.1.1970). |
| startupTimeNanos | Server startup time (nanoseconds part). |
| serverAddressIPv6 | Server IPv6 address (or IPv6 encoded IPv4 address). |
| serverPort | Server port (where server TCP/IP is listening). |
| serverStatusIF | Optional server status Field description. |
| serverStatus | Optional server data. |

Beacon message members.

A beacon from yet unknown serverAdddressIPv6:serverPort means new server. A beacon with the same serverAdddressIPv6:serverPort address and different startupTime{Seconds,Nanos} means that server was restarted.

NOTE: it was agreed that servers would have time set on every restart (via NTP).

NOTE: beacons are no logger used to report server-alive status (this is done by monitoring TCP/IP traffic – if there is traffic from server, then server is alive). This reduces UDP traffic. Server must at least send a couple of beacons to notify that it is alive (e.g. 1Hz). After this it can stop sending them, or does it with a low rate (one per every couple of minutes) to report serverStatus.

#### Connection validation (0x01)

The first message sent from the server to a client when TCP/IP connection is established. The message indicates that the server is ready to receive requests; the client must not send any messages on the connection until it has received the validate connection message from the server.

The purpose of the validate connection message is two-fold:

* It informs the client of the connection and protocol details.
* It prevents the client from writing a request message to its local transport buffers until after the server has acknowledged that it can actually process the request. This avoids a race condition caused by the server's TCP/IP stack accepting connections in its backlog while the server is in the process of shutting down: if the client were to send a request in this situation, the request would be lost but the client could not safely re-issue the request because that might violate at-most-once semantics.

The validate connection message guarantees that a server is not in the middle of shutting down when the server's TCP/IP stack accepts an incoming connection and so avoids the race condition.

struct connectionValidationRequestToClient {

int serverReceiveBufferSize;

int serverReceiveSocketBufferSize;

};

struct connectionValidationResponseFromClient {

int clientReceiveBufferSize;

int clientReceiveSocketBufferSize;

//TODO short priority;

};

#### Echo (0x02)

Echo diagnostic message. Usually sent when there is a suspicion that server is no longer functional (no TCP/IP traffic). Can also be send over UDP/IP (to get list of all servers).

struct echoRequest {

byte[] somePayload;

};

struct echoResponse {

byte[] samePayloadAsInRequest;

};

#### Search message (0x03)

Channel search message. Flow control needs to be implemented when send over UDP/IP.

struct searchRequest {

int searchSequenceId;

byte qosCode; // == boolean for “response required” (if none found)

struct { // not serialized as pvData

int clientChannelID;

string channelName;

} channels[];

};

Response is sent as messageType 0x04 (to avoid self-searching).

#### Search response (0x04)

Response to Search request (0x03) message.

struct searchResponse {

int searchSequenceId;

boolean found;

byte[16] serverAddressIPv6;

short serverPort;

int[] clientChannelIDs;

};

#### Create channel (0x07)

struct createChannelRequest {

struct { // not serialized as pvData

int clientChannelID;

string channelName;

} channels[];

};

struct createChannelResponse { // per channel

int clientChannelID;

int serverChannelID;

Status status;

//TODO short accessRights;

};

#### Destroy channel (0x08)

struct destroyChannelRequest {

struct { // not serialized as pvData

int clientChannelID;

int serverChannelID;

} channels[];

};

#### Channel get (0x0A)

struct channelGetRequestInit {

int serverChannelID;

int requestID;

byte qos = 0x08;

FieldDesc pvRequestIF;

PVField pvRequest;

};

struct channelGetResponseInit {

int requestID;

byte qos; // same as in request

Status status;

FieldDesc pvStructureIF;

};

struct channelGetRequest {

int serverChannelID;

int requestID;

byte qos = 0x40 mask for GET; 0x10 mask for DESTROY;

};

struct channelGetResponse {

int requestID;

byte qos; // same as in request

Status status;

BitSet changedBitSet;

PVField pvStructureData;

};

#### Channel put (0x0B)

struct channelPutRequestInit {

int serverChannelID;

int requestID;

byte qos = 0x08;

FieldDesc pvRequestIF;

PVField pvRequest;

};

struct channelPutResponseInit {

int requestID;

byte qos; // same as in request

Status status;

FieldDesc pvPutStructureIF;

};

struct channelPutRequest {

int serverChannelID;

int requestID;

byte qos = 0x00 mask for PUT; 0x10 mask for DESTROY;

BitSet toPutBitSet;

PVField pvPutStructureData;

};

struct channelPutResponse {

int requestID;

byte qos; // same as in request

Status status;

};

struct channelGetRequest {

int serverChannelID;

int requestID;

byte qos = 0x40;

};

struct channelGetResponse {

int requestID;

byte qos; // same as in request

Status status;

PVField pvPutStructureData;

};

#### Channel put-get (0x0C)

struct channelPutGetRequestInit {

int serverChannelID;

int requestID;

byte qos = 0x08;

FieldDesc pvRequestIF;

PVField pvRequest;

};

struct channelPutGetResponseInit {

int requestID;

byte qos; // same as in request

Status status;

FieldDesc pvPutStructureIF;

FieldDesc pvGetStructureIF;

};

struct channelPutGetRequest {

int serverChannelID;

int requestID;

byte qos = 0x00 mask for PUT\_GET; 0x10 mask for DESTROY;

BitSet toPutBitSet;

PVField pvPutStructureData;

};

struct channelPutGetResponse {

int requestID;

byte qos; // same as in request

Status status;

PVField pvGetStructureData;

};

// get remote put structure data

struct channelGetPutRequest {

int serverChannelID;

int requestID;

byte qos = 0x80;

};

struct channelGetPutResponse {

int requestID;

byte qos; // same as in request

Status status;

PVField pvPutStructureData;

};

// get remote get structure data

struct channelGetGetRequest {

int serverChannelID;

int requestID;

byte qos = 0x40;

};

struct channelGetGetResponse {

int requestID;

byte qos; // same as in request

Status status;

PVField pvGetStructureData;

};

#### Channel monitor (0x0D)

struct channelMonitorRequestInit {

int serverChannelID;

int requestID;

byte qos = 0x08;

FieldDesc pvRequestIF;

PVField pvRequest;

};

struct channelMonitorResponseInit {

int requestID;

byte qos; // same as in request

Status status;

FieldDesc pvStructureIF;

};

struct channelStartMonitorRequest {

int serverChannelID;

int requestID;

byte qos = 0x44;

};

struct channelStopMonitorRequest {

int serverChannelID;

int requestID;

byte qos = 0x04;

};

struct channelDestroyMonitorRequest {

int serverChannelID;

int requestID;

byte qos = 0x10;

};

struct channelMonitorResponse {

int requestID;

byte qos; // same as in request

BitSet changedBitSet;

PVField pvStructureData;

BitSet overrunBitSet;

};

#### Channel array (0x0E)

struct channelArrayRequestInit {

int serverChannelID;

int requestID;

byte qos = 0x08;

FieldDesc pvRequestIF;

PVField pvRequest;

};

struct channelArrayResponseInit {

int requestID;

byte qos; // same as in request

Status status;

FieldDesc pvArrayIF;

};

struct channelGetArrayRequest {

int serverChannelID;

int requestID;

byte qos = 0x40 mask for GET; 0x10 mask for DESTROY;

size offset; // -1 == unspecified

size count; // -1 == unspecified

};

struct channelGetArrayResponse {

int requestID;

byte qos; // same as in request

Status status;

PVField pvArrayData;

};

struct channelPutArrayRequest {

int serverChannelID;

int requestID;

byte qos = 0x00 mask for PUT; 0x10 mask for DESTROY;

size offset; // -1 == unspecified

PVField pvArrayData;

};

struct channelPutArrayResponse {

int requestID;

byte qos; // same as in request

Status status;

};

struct channelSetLengthRequest {

int serverChannelID;

int requestID;

byte qos = 0x80 mask for SET\_LENGTH; 0x10 mask for DESTROY;

size offset; // -1 == unspecified

size count; // -1 == unspecified

};

struct channelSetLengthResponse {

int requestID;

byte qos; // same as in request

Status status;

};

#### Channel cancel request (0xF)

// destroys any request with given requestID

struct channelDestroyRequestInit {

int serverChannelID;

int requestID;

};

#### Channel process (0x10)

struct channelProcessRequestInit {

int serverChannelID;

int requestID;

byte qos = 0x08;

FieldDesc pvRequestIF;

PVField pvRequest;

};

struct channelProcessResponseInit {

int requestID;

byte qos; // same as in request

Status status;

};

struct channelProcessRequest {

int serverChannelID;

int requestID;

byte qos = 0x00 mask for PROCESS; 0x10 mask for DESTROY;

};

struct channelProcessResponse {

int requestID;

byte qos; // same as in request

Status status;

};

#### Channel get field (0x11)

struct channelGetFieldRequest {

int serverChannelID;

int requestID;

string subFieldName; // entire record if empty

};

struct channelGetFieldResponse {

int requestID;

Status status;

[if status is success] FieldDesc subFieldIF;

};

#### Message (0x12)

struct message {

int requestID;

byte messageType; // info = 0, warning = 1, error = 2, fatalError = 3

string message;

};

#### Channel RPC (0x14)

struct channelRPCRequestInit {

int serverChannelID;

int requestID;

byte qos = 0x08;

FieldDesc pvRequestIF;

PVField pvRequest;

};

struct channelRPCResponseInit {

int requestID;

byte qos; // same as in request

Status status;

FieldDesc pvStructureIF;

};

struct channelRPCRequest {

int serverChannelID;

int requestID;

byte qos = 0x00 mask for RPC; 0x10 mask for DESTROY;

BitSet toPutBitSet;

PVField pvStructureData;

};

struct channelRPCResponse {

int requestID;

byte qos; // same as in request

Status status;

FieldDesc pvResponseIF;

PVField pvResponseData;

};

### Example of pvAccess communication

