technicolor

TECHNICOLOR DDS USER MANUAL

Draft Version 15:32:33, 31 August 2012

TECHNICOLOR DDS USER MANUAL

Draft Version 15:32:33, 31 August 2012

1 Overview

The purpose of this document is to make the user familiar with working with the Technicolor DDS Middleware implementation.

Technicolor DDS provides a Middleware approach that minimizes the memory footprint of DDS-applications, without sacrificing their speeds, in order to make the DDS Publish/Subscribe methodology usable on low-memory, low-performance embedded devices.

The next chapters will give a small introduction in the Middleware concept and how the Technicolor DDS can make it easier for application writers to use DDS.

The Technicolor DDS Data Type system will be described in detail, since this is one of the most important changes in Technicolor DDS compared to other DDS implementations.

The document will also give some guidelines to application writers which kind of DDS data eventing approaches are available, and which is more appropriate in what situation, e.g. using either Listener functions, or using a WaitSet-based approach.

A number of environment variables are available in Technicolor DDS that can be used to influence the usage of Multicast and of Interface IP addresses. These environment variables will also be discussed.

A drastic per-node DDS memory reduction is possible using the Technicolor Central Discovery Daemon.

For a good look into how applications are actually using their DDS resources, the Technicolor DDS implementation can provide lots of ways to examine this in detail. Some of those ways are by default compiled out in order to minimize the code size, but it is quite easy to enable more exhaustive debugging features. The available compilation options will be explained in detail.

We will also explain how to optimise memory usage even more using the Technicolor DDS memory pool system.

A number of DDS example programs are provided which are quite useful on their own, such as the Shapes demo program, and the Bandwidth and Latency test programs. These will also be explained.

2 Middleware concepts and DDS

Topics:

- *Need for a middleware*
- *Publish-subscribe methodology*
- *DDS in a nutshell*
- *Technicolor DDS*
- *DDS Architecture*

Need for a middleware

Software applications are becoming increasingly distributed nowadays. Whereas previously standalone application programs were sufficient for most purposes, this is no longer the case.

However, making a distributed application, even though it allows a much more powerful systems, has an associated cost (nothing ever comes for free).

We can envisage a distributed application on several data distribution levels:

- Multiple threads within a process
- Multiple processes within a single node
- Multiple processes in distinct nodes.

For each of these approaches there are advantages and disadvantages.

The first level, i.e. multiple threads communicating on, typically, shared data is one of the easier approaches, when using a multi-threading library such as pthreads. There are, however a number of problems associated with this:

- A misbehaving thread can impact other well-behaving threads, making the resulting system a lot less robust than expected.
- It does not scale well, and leads to huge system components.
- Makes it very hard to pull apart the different threads into different processes in a later stage, since the possibility of data sharing is typically used in unexpected situations by the application writers.

For the second level of data distribution, e.g. multiple processes within a single node, there are also some mechanisms, out of the box available on many systems, such as dbus. This mechanism typically works very well within a single node, but it doesn't scale when an application outgrows a single node, and when it becomes necessary to use multiple heterogenous nodes communicating together. Take the example of a Technicolor Gateway, a Technicolor Settop box and some management software running on a management PC. In this case, you automatically end up with multiple heterogenous nodes, all needing to communicate with each other.

The third level of distribution is the most general but leads to a lot more complex communication mechanisms. The communication overhead then becomes a heavy burden on application writers, since they need to be constantly aware of it while developing their software.

The idea of the middleware now, is to shield the details of the communication and the communication overhead from the application writers. Which data distribution level is used, then becomes a last-minute build decision, instead of having this visible on application level. Application writers simply use a standardized way of sharing their data with others, not knowing where the other application components are located, be it in the same process, another process on the same node, or on an entirely different node altogether. Not having to deal with the communication then allows application writers to focus much more on the usage of their data, which is a lot more important to them.

Publish-subscribe methodology

Various types of distributed middlewares exist, based on different approaches into how end-to-end communication occurs:

- Client-server. This works well when all data is centralized, such as in databases, transaction processing systems and in file servers. In this approach, a central server handles client requests and sends responses when ready. Typically this doesn't scale well, since the server is a central point of failure and quickly becomes a bottleneck in a system. Also, when multiple nodes are able to generate data, they all have to sent this data to the server before it can be redistributed.
- Message-Passing. The message-passing approach extends the client-server system to a more distributed topology. It uses queues of messages to individual processes, which then process a single message at the time and responding to them if appropriate. Message passing is a lot more efficient than client-server, since it allows peer-to-peer communication. However, there are also a number of problems associated with this:
    1. The message passing approach is not data-centric, and requires the application to know the details as to 'where' the data is available, and then asking the appropriate data handling process for the data it is interested in.
    2. If providing processes become unavailable, applications need to know this and must recover properly, which is often quite complex to do.
- Publish-subscribe The Publish-subscribe approach makes communication a lot easier, since it adds a data-centric layer to the message passing approach. In this way, it is no longer required for the application to know where the data is located. It just subscribes to a specific data topic and will be notified when this data is available, is updated and is removed. Data publishers also don't need to listen to requests for data. They simply publish the data and the middleware will take care of proper distribution between the publishers and the subscribers.

The publish-subscribe approach is clearly the most powerful and most application-friendly approach to sharing data. Since it is fully peer-to-peer, it is also very fast and scales very well to a reasonable amount of distinct components (a hundred components seems feasible with modest memory requirements). Its design also makes it a lot easier to recover on various error situations, such as:

- Components disappearing due to crashes, and restarting.
- Components starting up in random order.

A number of publish-subscribe middlewares exist currently. The most widely known are Java Messaging Service (JMS) and Data Distribution System (DDS). The main differences between the two are:

- JMS is a lot slower than DDS, but allows both publish-subscribe and message-passing.
- DDS was designed from the start for real-time systems, and is much better suited for this.
- DDS has no builtin message-passing mechanism, although it is not that difficult to do this on top. In fact, the Remote Message Invocation (RMI) extension to DDS is a recent development, which is doing exactly this.

DDS in a nutshell

So what exactly does DDS deliver to applications?

The DDS publish-subscribe model connects information producers (publishers) with information consumers (subscribers).

The overall distributed application is typically composed of processes called "participants," each running in a separate address space, possibly on different computers and in different execution environments. A DDS participant may simultaneously publish and subscribe to a number of typed data 'topics'. Type safety is an important aspect in DDS and the data interface to and from the applications is therefore strongly typed.

This strong typing is achieved by describing topic data types using a data definition language that closely resembles the familiar C/C++ data definition notations, but is language independent (IDL file). Later on, an off-line tool allows generation of typecode data and type-specific functions appropriate for a specific destination execution environment.

This has a number of benefits:

1. Since DDS knows about the data types, it can not only take care of the differences between little- and big-endian machines, but it can thereby overcome the different data representations between multiple execution environments, such as C, C++, Java, etc.
2. Strongly typed data interfaces are generated, making the DDS API for applications less error-prone.
3. Data types can be "keyed" by marking specific fields in the type as "key" fields, so that multiple instances of data, e.g. multiple data streams can be possible, identified by its "key".

The DDS standard defines a communications relationship between the various publishers and subscribers. The data communications are decoupled in space (nodes can be anywhere), time (delivery may be immediately after publication or later), and flow (delivery may be reliably or best-effort and made at controlled bandwidths).

To increase scalability, topics may contain multiple independent data channels identified by "keys". This allows nodes to subscribe to many, possibly thousands, of similar data streams with a single subscription. When the data arrives, the middleware can sort it by the key and deliver it for efficient processing.

There is no requirement for a one-to-one relationship between publishers and subscribers. On the contrary, an any-to-any relationship is the overall model.

The DDS standard is fundamentally designed to work over unreliable transports, such as UDP or wireless networks. Efficient, direct, peer-to-peer communications, or multicasting, will be used to optimize data transfers, reliable (even over multicast) or best-effort.

These communication aspects are typically specified by the application when subscribing to data topics. This is done with optional so-called Quality of Service parameters.

Filtering of received data samples is possible, either on data contents (using SQL-like expressions) or by specifying the requested rate of received data. Whether this filtering occurs on the publisher side or on the subscriber side depends on the DDS implementation. Good DDS implementations are able to do both.

Technicolor DDS

As described before, the publish-subscribe way of exchanging data between components is a very powerful idea.

Of course, an idea is one thing, but an efficient implementation is another thing altogether.

While examining DDS as a technology, it became apparent that current commercial DDS implementations had huge memory and performance requirements, making them unfit for use on embedded platforms. It also was not clear why exactly this seemed to be the case, since DDS does (from our point of view) allow very efficient and memory-constrained implementations.

The focus of DDS vendors at that time was (not yet) intended for small platforms, but was instead mainly driven by use cases like:

- High-speed financial transaction processing.
- Avionics industry
- Defense industry

Since most of these do not have tight memory requirements, although they do have high performance requirements, this was not seen as a big issue.

Also, the speed aspects of DDS were seen as prevalent to resource usage. Optimizing an implementation for speed typically sacrifices resources due to strategies such as heavy caching, and overly preallocating receive buffers.

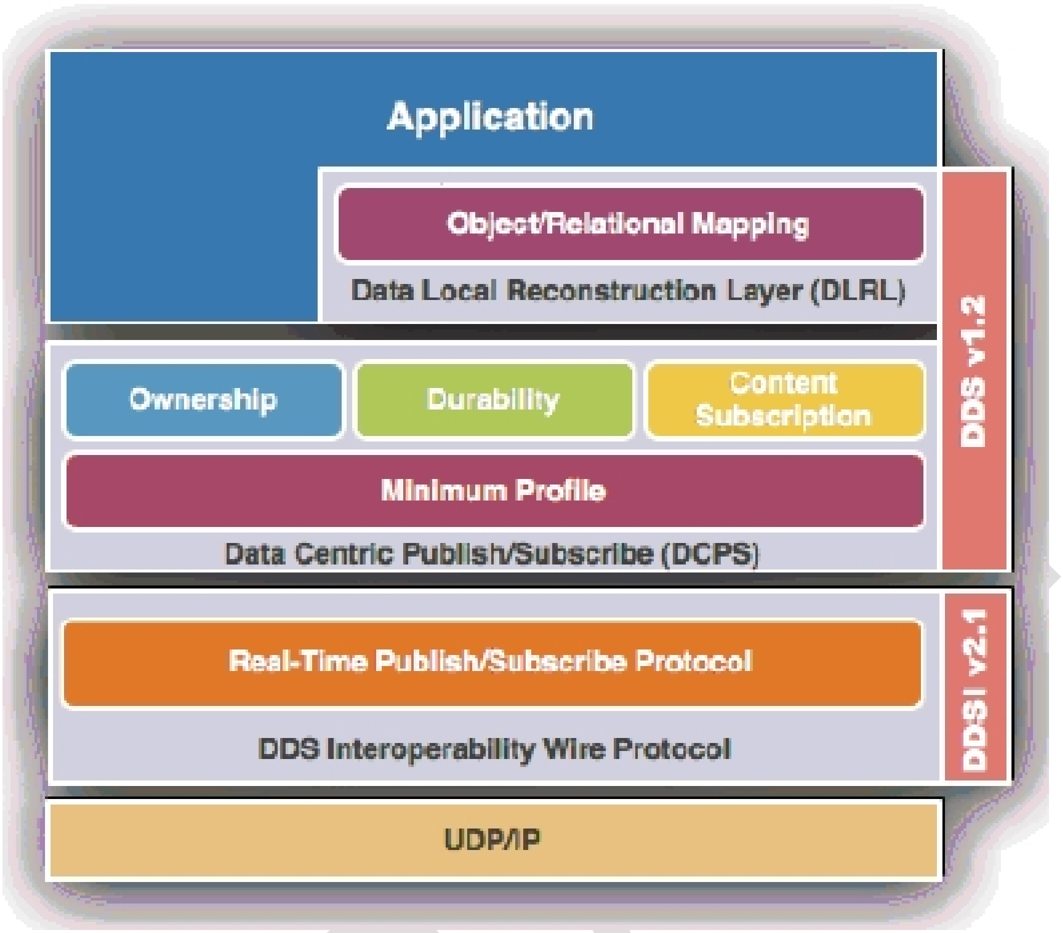
The result of all this is that it became necessary for Technicolor to build their own DDS implementation in order for DDS to be optimally usable on the current Gateway and Settop box platforms.

The current version of Technicolor DDS runs on multiple Operating Systems, i.e. Unix, Linux and Windows and is able to run and is actively supported on x86 platforms in 32-bit as well as in 64-bit mode. It has also been tested on ARM-based Android platforms, and is used on gateways based on a big-endian MIPS processor.

Currently, Technicolor DDS is an integral part of the Rebus Component Development framework, and as such forms the basis of systems using the Core Middleware Framework.

DDS Architecture

As can be seen in the figure below, a DDS middleware is structured in a number of layers:


Application
Object/Relational Mapping
Data Local Reconstruction Layer (DLRL)
Ownership
Durability
Content Subscription
Minimum Profile
Data Centric Publish/Subscribe (DCPS)
DDS v1.2
Real-Time Publish/Subscribe Protocol
DDS Interoperability Wire Protocol
DDSI v2.1
UDP/IP

The DDS specification gives a choice of 2 distinct APIs to the user, allthough most DDS vendors implement only one of those APIs.

The most common API, which every DDS vendor currently supports, is called the Data Centric Publish Subscribe (DCPS) API, which is the most basic of the two.

The second API is the Data Local Reconstruction Layer (DLRL) API, an optional layer on top of DCPS, which is a lot more Object-Oriented. It allows applications to describe objects with methods as well as attributes, which are either local or shared. DLRL does not do RPCs however, all methods are local.

Since only a few DDS vendors support DLRL, using this API would lead to interoperability issues with different DDS vendors. Also, because this requires an additional software layer on top of the DCPS API with additional memory and performance requirements, Technicolor DDS has chosen not to support this API.

For both APIs, the OMG standard gives a description in a language-independent format (IDL) in order to allow multiple execution environments and also to allow multiple programming languages. Due to this choice, the conversion from this format to an effective programming language API still leaves some design choices open. A disadvantage of this approach thus means that a specific programming language DCPS API may differ slightly from one DDS vendor to the next.

In order to avoid these minor differences, and also because for object-oriented languages it is possible to wrap the DCPS API into language-specific class-based APIs which are of course a lot easier to work with, OMG has standardized some specific programming language specifications for the DCPS API. Specifications are defined for both C++ and Java.

Currently, the DCPS API for Technicolor DDS is only available for the C programming language, allthough we do envisage C++ and Java APIs in the near future.

The DDS specification allows many different types of data durability, from volatile up to persistent durability. The first will remove data as fast as possible from the data caches. The persistent durability stores data automatically on a storage medium (such as a disk) without the users intervention. Persistent durability is currently not implemented by Technicolor DDS.

For effective communication over a network, an additional software layer below DCPS is responsible This layer, which is called the Real-Rime Publish Subscribe (RTPS) layer is able to exchange data between DDS components in a standardized manner, providing necessary compatibility between different DDS vendors.

RTPS is responsible for coping with the differences between different nodes (big/little endianess) and between subtle different data layouts between different execution environments. This is achieved in practice by marshalling local data representations into a network-specific format, which is execution environment agnostic, before sending the data on the wire. On reception of messages, the data is unmarshalled to the local specific execution environment data representation.

Since RTPS is designed to operate over multiple networking protocols, there is no requirement for a specific networking protocol, and DDS thus allows both operation over local LANs as well as over Wide Area Networks (WAN), typically using different protocols.

Only the interface between RTPS and UDP/IP is defined in the DDS standards, but in practice, many vendors offer support for IPv6, secure transport over TCP, and even shared-memory transports within device nodes.

Technicolor DDS has support for RTPS over UDP/IPv4 by default. Support for RTPS over UDP/IPv6 is optional but it can easily be added at compile-time when the -DDDS_IPV6 compile option is used.

In the latter case, it can be controlled via environment variables as to which IP variant (or both) are to be used. When both are used simultaneously, it is also selectable which protocol has precedence when either can be used, and which of the protocols will have discovery enabled.

Other transport protocols are not in the current scope.

3 Technicolor DDS features

Topics:

- *Design goals*
- *Centralized Discovery Daemon*
- *Technicolor DDS Debug features*

Design goals

Due to Technicolor DDS being used on platforms having severe memory footprint restrictions, and also because it is an intrinsic part of the Technicolor core middleware, which comprises more than DDS alone, and instead provides a full component development framework, the design of Technicolor DDS is slightly different from other commercial DDS implementations.

Following design goals were envisaged:

- Memory usage needs to be kept as low as possible, both for core data structures where DDS metadata must be stored in the most optimal manner, as well as for real data buffers, storing topic data.
- Never create an entity until it is really needed to have it.
  1. RTPS Reader/Writers are only created when a remote entity is detected that has an interest in the topic data.
  2. Data caches can autonomously transfer data between a local Writer and a local Reader without the need for RTPS.
  3. Discovery data readers are only created when the user expresses an interest in it.
- When possible, reuse the same data (sometimes determined by hashes), using reference counts. This is done, for example, for:
  1. Stored user data, which is shared between DDS history caches.
  2. Locator structures.
  3. Discovery data.
  4. QoS parameter sets.
  5. Character strings are stored only once.
- Using memory pools instead of ´pure´ malloc()/free() has several advantages, such as:
  1. Severely limiting allocation overhead and memory fragmentation.
  2. Allocation/free of data chunks is much faster.
  3. Allows to specify upfront what the pool limits may be.
  4. The application´s memory usage can be monitored due to the pool´s bookkeeping.
- Contrary to other DDS implementations, data type marshalling and unmarshalling is data-driven instead of function-driven (see TSM type specification).
- Data types are generated primarily from Rebus layer XML-based data, which is an additional core middleware API on top of Technicolor DDS. This does not preclude an IDL data definition compiler, it just isn´t ready yet, and for now we rely on generated type information from Rebus, or manually created.
- Since the Rebus layer is a C-component, there is no need (yet) for other execution environments, allthough we also envision other execution environments (C++ and Java), this is of lesser importance at the moment.
- Focus on easy debugging of application as well as Technicolor DDS operations.
- Use a node-centralized Discovery data manager, the Central Discovery Daemon (CDD), in order to prevent components of learning topics via the DDS Discovery mechanisms in which they are not interested.

Centralized Discovery Daemon

Ordinarily, DDS implementations learn all topics that are present in the DDS domains in which they are interested, even though they usually are only interested in a subset of the collective Topic data.

This is done, by design, because it is not known initially which topics will be referenced in the future by applications. It could be, for example, that Topics are discovered first, after which they might be referenced by local Readers or Writers. A strategy that simply throws away Discovered data for which there is no local Topic yet, will not work at all. The data that was thrown out will not come back automatically afterwards, just like that.

The solution to this problem is a node-central Discovery manager, called the CDD component. The Centralized Discovery Daemon is a stand-alone (daemonized) program that allows to optimize this per-node Discovery data so that discovered data does not need to be kept in components that are not interested in this data.

The CDD, when active, allows to communicate the state of Discovery data to other Technicolor DDS components. This way, those DDS components will be informed of matching discovery data the moment they subscribe to Topics and there are already 'early joiners' for this topic.

Technicolor DDS Debug features

The Technicolor DDS implementation has the following features that are useful for debugging user- as well as DDS-specific code and data:

1. A debug command handler (shell) can easily be attached to application code.
2. An application debugger (gdb, ddd, etc.) can 'call' Technicolor DDS debug commands.
3. Any Technicolor DDS application can optionally be verified/monitored remotely using telnet to a built-in debug server giving access to the Debug shell.
4. The Debug shell can give detailed information on memory usage, created as well as discovered DDS entities, and it allows detailed inspection of History Cache data.
5. RTPS message tracing can be enabled for any topic, even the builtin Discovery topics.

4 Data Types and Topics

Topics:

Introduction

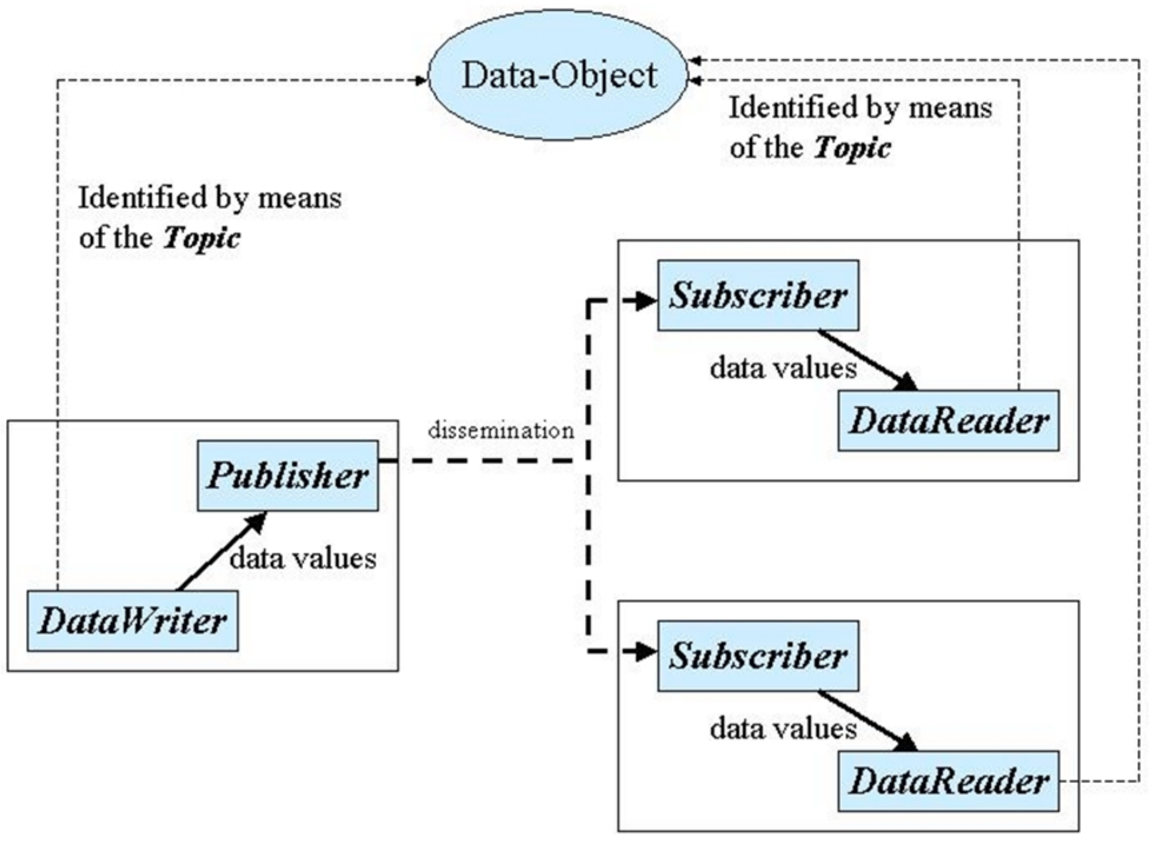
As explained before, data is transferred between Data Writers and Data Readers using properly typed writer and reader functions, based on an intimate knowledge of the exact data types that are used.

This is done using the concept of topics and types, where a DDS topic corresponds to a single data type. However, several topics may refer to the same data type.

Topics may range from a single instance to a whole collection of instances of the given type. Different instances must be distinguishable. This is achieved by means of the value of some data fields of the objects that form the key to that data set.

The exact type of a topic and the fields it contains can either be constructed dynamically (using the Dynamic Type system), or can be specified statically using external tools . In the latter case, a descriptive language such as IDL or XML is used to define the data contents. From this, the code/data that is needed to handle the Topic data and the on-the-wire encoding/decoding logic is generated.

See the figure below for the relationship between Writers, Readers, Topics and types:



Types that can be specified are:

- Integer types (16-bit, 32-bit, 64-bit).
- Unsigned types (16-bit, 32-bit, 64-bit).

- Floating point types (32-bit, 64-bit, 128-bit)[1]
- Fixes point types (up to 31 digits).[2]
- Bytes
- Booleans
- Chars (8-bit and 32-bit) - Allows UTF-8, UTF-16 and UTF-32.
- Bitsets
- Enumerations
- Alias
- Collections (Map, Sequence, Array and String).
- Aggregations (Union and Struct)

Some metadata is supported as well:

- Annotations
- Verbatim text

The definition of which types can be expressed has been extended recently with an additional DDS specification, called X-types, or the Extended Types specification, which has added support for the Map type, Annotations and Verbatim text. Originally these features were not available. The X-types specification has also added support for sparse data structures and evolution of data types, as well as providing additional marshalling methods, and an additional API for dynamically adding types and for dynamically accessing data. Introspection of types, locally generated or received from peers is also possible.

Technicolor DDS properly supports the extensions that are described in the X-types specification, as will be demonstrated further.

Technicolor DDS Type Support

In Technicolor DDS, static types are created from an intermediate type description format, which is a C-data structure.

This type description can be generated in different ways:

- Created from the Rebus component framework, which is a layer on top of Technicolor DDS, and which provides tools to generate this type description from an XML-based service description.
- A tool is available that can be used to generate the intermediate type from an IDL-based type specification.
- It can be hand-written, although the preferred way is of course one of the tools described above.

The intermediate type data is called the TSM format and has the following layout (as defined in dds/dds_types.h):

```
struct dds_typesupport_meta_st {
      CDR_TypeCode_t             tc;     /* Type code. */
      tsm_flags                  flags;  /* Key field, Dynamic data, ... */
```

[1] Technicolor DDS only supports 128-bit floating point numbers when compiled with a special preprocessor definition.

[2] Not currently supported in Technicolor DDS.

```
      const char                  *name;  /* Name of the field (info). */
      size_t                      size;   /* Size of the corresponding
                                             container or C string. */
      size_t                      offset; /* Offset of the field in its
                                             corresponding container. */
      unsigned int                nelem;  /* Number of elements in the
                                             container. */
      int                         label;  /* The label of a union member. */
      const DDS_TypeSupport_meta *tsm;    /* When TYPEREF is instantiated. */
};
```

The actual data in this structure depends on the type code id, which can have the following possible values as described below:

```
typedef enum {
      CDR_TYPECODE_SHORT = 1,     /* 16-bit signed integer. */
      CDR_TYPECODE_USHORT,        /* 16-bit unsigned integer. */
      CDR_TYPECODE_LONG,          /* 32-bit signed integer. */
      CDR_TYPECODE_ULONG,         /* 32-bit unsigned integer. */
      CDR_TYPECODE_LONGLONG,      /* 64-bit signed integer. */
      CDR_TYPECODE_ULONGLONG,     /* 64-bit unsigned integer. */
      CDR_TYPECODE_FLOAT,         /* 32-bit floating point number. */
      CDR_TYPECODE_DOUBLE,        /* 64-bit floating point number. */
#ifdef LONGDOUBLE
      CDR_TYPECODE_LONGDOUBLE,    /* 128-bit floating point number. */
#endif
      CDR_TYPECODE_FIXED,         /* Fixed-point number. */
      CDR_TYPECODE_BOOLEAN,       /* Boolean flag. */
      CDR_TYPECODE_CHAR,          /* 8-bit character. */
      CDR_TYPECODE_WCHAR,         /* 32-bit wide character. */
      CDR_TYPECODE_OCTET,         /* 8-bit byte. */
      CDR_TYPECODE_CSTRING,       /* 8-bit character string. */
      CDR_TYPECODE_WSTRING,       /* 32-bit wide character string. */
      CDR_TYPECODE_STRUCT,        /* C-struct. */
      CDR_TYPECODE_UNION,         /* C-union. */
      CDR_TYPECODE_ENUM,          /* C enumeration type. */
      CDR_TYPECODE_SEQUENCE,      /* Variable length element array. */
      CDR_TYPECODE_ARRAY,         /* Fixed length element array. */
      CDR_TYPECODE_TYPEREF        /* Reference to another type. */
} CDR_TypeCode_t;
```

The flags field currently has the following valid bits:

```
#define TSMFLAG_KEY      1    /* Is a key field. */
#define TSMFLAG_DYNAMIC  2    /* Container structure has dynamic data. */
#define TSMFLAG_MUTABLE  4    /* Extra container fields can be added later. */
#define TSMFLAG_OPTIONAL 8    /* Field is optional, i.e. is a pointer field that
                                 can be skipped. */
#define TSMFLAG_SHARED   16   /* Field is a pointer field instead of a real data
                                 field. */
```

A constructed DDS type must always start with a *struct* TSM, which has additional TSMs directly following the first for each individual field of the structure.

The significance of the TSM fields is described in the following table, depending on the type code:

Table 1: TSM fields

tc	flags	name	size	offset	nelem	label	tsm
SHORT	KEY	char pointer to name[3].	-	Container offset in bytes.	-	-	-
USHORT	KEY	*idem*	-	*idem*	-	-	-
LONG	KEY	*idem*	-	*idem*	-	Enum value (or not significant).	-
ULONG	KEY	*idem*	-	*idem*	-	-	-
LONGLONG	KEY	*idem*	-	*idem*	-	-	-
ULONGLONG	KEY	*idem*	-	*idem*	-	-	-
FLOAT	KEY	*idem*	-	*idem*	-	-	-
DOUBLE	KEY	*idem*	-	*idem*	-	-	-
LONGDOUBLE	KEY	*idem*	-	*idem*	-	-	-
FIXED	KEY	*idem*	-	*idem*	-	-	-
BOOLEAN	KEY	*idem*	-	*idem*	-	-	-
CHAR	KEY	*idem*	-	*idem*	-	-	-
WCHAR	KEY	*idem*	-	*idem*	-	-	-
OCTET	KEY	*idem*	-	*idem*	-	-	-
CSTRING	KEY, DYNAMIC	*idem*	-	*idem*	C-string size or 0 if dynamic (char *)	-	-
WSTRING	KEY, DYNAMIC	*idem*	-	*idem*	C-wstring size or 0 if dynamic (wchar *)	-	-
STRUCT	KEY, DYNAMIC	*idem*	Total C-struct data size in bytes.	*idem*	Number of struct fields (that follow as TSMs).	-	-

[3] Although the type name is only really required for the top-level TSM-struct, the name field will also be used when SQL content filter specifications refer to these fields, in order to find the proper data offset in received data samples.

tc	flags	name	size	offset	nelem	label	tsm
UNION	KEY, DYNAMIC	*idem*	Total C-union data size in bytes	*idem*	Number of union fields (that follow as TSMs).	-	-
ENUM	KEY	*idem*	-	*idem*	Number of enum elements (which follows as LONG TSMs).	-	-
SEQUENCE	-	*idem*	-	*idem*	Maximum number of sequence elements or 0. Sequence type follows as TSM.	-	-
ARRAY	KEY, DYNAMIC	*idem*	Total C-array size in bytes.	*idem*	Number of array elements (type follows as additional TSM).	-	-
TYPEREF	KEY	*idem*	-	*idem*	-	-	Points to referred type.

Notes on the flags field.

Typically C-structs are allowed to embed substructures. This can easily be expressed with the TSM mechanism by specifying fields of a CDR_TYPECODE_STRUCT type, which are then followed again by structure field definitions in a recursive manner.

The handling of the flags field in embedded structs as well as for CDR_TYPECODE_TYPEREF referenced structs can become somewhat complex due to this. The following rules are therefore used in Technicolor DDS to ensure a consistent behavior:

- If a struct reference has the KEY attribute, but none of its fields has the KEY attribute, this KEY attribute will automatically be applied to all structure fields, as if each field had it set specifically.
- If fields have the KEY attribute, but the struct reference doesn't, no fields will be used as keys. The parent structure must thus always have the KEY attribute set in order for any of its member fields to have this also.
- The DYNAMIC attribute **must** be set for all structure fields that have pointer data, such as dynamic strings (having a 0-length, i.e. indicating *char* *) and sequences.

- The DYNAMIC attribute must be properly set for parents. I.e. if any field is dynamic, the parent structure must have the DYNAMIC attribute as well, and this must propagate up to the highest level container struct.

Registering a type in DDS

Once a type is created as a list of TSMs, it can be used as the foundation of a new Technicolor DDS type. This can be done using the *DDS_DynamicType_register()* function as follows:

```
DDS_TypeSupport *DDS_DynamicType_register(const DDS_TypeSupport_meta *tc);
```

Where tc is a pointer to the first TSM in the list.

When this function is successful, i.e. it has managed to convert the TSM list to the type representation used internally in Technicolor DDS, a new type is returned that can be used for registering the type within the DDS domains. If not successful, the function returns a NULL- result.

For each domain where the type is to be used, the following function should to be called:

```
DDS_ReturnCode_t DDS_DomainParticipant_register_type(
        DDS_DomainParticipant self,
        const DDS_TypeSupport *ts,
        const char *type_name
);
```

Where self is the DomainParticipant which should have been created previously, ts is the TypeSupport pointer that was registered via *DDS_DynamicType_register()*, and type_name is the name of the type within the domain.

If something went wrong with registering the type, a standard DDS error code will be returned.

This function may be called multiple times, in case the type is needed in multiple domains.

If a type is no longer needed in a domain, the *DDS_DomainParticipant_unregister_type()* function can be used. This is not really needed, however, since this unregister function is used implicitly for each registered type while cleaning up DomainParticipants, when the *DDS_DomainParticipant_delete_contained_entities()* function is used.

If the type is no longer needed in any DDS domain, the *DDS_DynamicType_free()* function can be called to free all internal type representation data.

Example Type usage

Shapes type.

A very simple example from the shapes demo as an IDL type definition:

```
const long MAXC = 127;

struct Shape_t {
    string<MAXC> color;   //@Key
    int          x;
    int          y;
    int          shapesize;
};
```

As a C-language type definition:

```
#define MAXC 127

typedef struct shape_st {
    char  color[MAXC+1];   //@Key
    int   x;
    int   y;
    int   shapesize;
} Shape_t;
```

This would be specified as a static type definition in TSM format as:

```
#include <stddef.h>

static DDS_TypeSupport_meta shape_tsm [] = {
    { CDR_TYPECODE_STRUCT, TSMFLAG_KEY, "Shape_t",
      sizeof (Shape_t), 0, 4, 0, NULL },
    { CDR_TYPECODE_CSTRING, TSMFLAG_KEY, "color",
      MAXC+1, offsetof (Shape_t, color), 0, 0, NULL },
    { CDR_TYPECODE_LONG, 0, "x",
      0, offsetof (Shape_t, x), 0, 0, NULL },
    { CDR_TYPECODE_LONG, 0, "y",
      0, offsetof (Shape_t, y), 0, 0, NULL },
    { CDR_TYPECODE_LONG, 0, "shapesize",
      0, offsetof (Shape_t, shapesize), 0, 0, NULL }
};
```

This conversion can be done either manually, or by using the IDL data compiler.

In order to use this type in a DDS client program, the type still needs to be converted to a DDS_TypeSupport type:

```
     DDS_TypeSupport *shape_ts;

     shape_ts = DDS_DynamicType_register (shape_tsm);
     if (!shape_ts)
          return (DDS_RETCODE_OUT_OF_RESOURCES);
```

This type representation can be registered in a DDS Domain with the following function:

```
     error = DDS_DomainParticipant_register_type (part, shape_ts, "ShapeType");
     if (error)
          return (error);
```

Sending data of this type could then be done as follows:

```
     ShapeType_t data;

     strcpy (data.color, "RED");
     data.x = 10;
     data.y = 50;
     data.shapesize = 25;
     error = DDS_DataWriter_write (w, &data, 0);
```

When a sample of this type is received, the following code might be used:

```
     error = DDS_DataReader_take (dr, &ssample, &sinfo, 1, ss, vs, is);
     if (!error && DDS_SEQ_LENGTH (sinfo)) {
          info = DDS_SEQ_ITEM (sinfo, 0);
          if (info->valid_data) {
               sample = DDS_SEQ_ITEM (ssample, 0);
               printf ("%s x=%u, y=%u, size=%u, sample->color,
                                                  sample->x, sample->y,
                                                  sample->shapesize);
          }
          DDS_DataReader_return_loan (dr, &ssample, &sinfo);
     }
```

Simple types and enums.

A slightly more complex example but using only simple types and an enumeration type:

First we show the IDL definition:

```
enum anenum2 {
    a,
    b,
    c
};

struct s1 {
    unsigned short u16;
    short i16;
    unsigned long u32;
    long i32;
    unsigned long long u64;
    long long i64;
    float fl;
    double d;
    char ch;
    bool b;
    octet o;
    anenum2 anenumelement;
};
```

The corresponding C-type:

```
#include <stdint.h>

typedef enum {
    a,
    b,
    c
} anenum2;

struct s1 {
    uint16_t u16;
    int16_t i16;
    uint32_t u32;
    int32_t i32;
    uint64_t u64;
    int64_t i64;
    float fl;
    double d;
    char ch;
    unsigned char b;
    unsigned char o;
    anenum2 anenumelement;
};
```

Resulting in the following TSM definitions[4]:

```
#include <stddef.h>

static DDS_TypeSupport_meta tsm1[] = {
    { .tc = CDR_TYPECODE_STRUCT, .name = "s1",
      .size = sizeof(struct struct1), .nelem = 12 },
    { .tc = CDR_TYPECODE_USHORT, .name = "u16",
      .offset = offsetof(struct s1, u16) },
    { .tc = CDR_TYPECODE_SHORT, .name = "i16",
      .offset = offsetof(struct s1, i16) },
    { .tc = CDR_TYPECODE_ULONG, .name = "u32",
      .offset = offsetof(struct s1, u32) },
    { .tc = CDR_TYPECODE_LONG, .name = "i32",
      .offset = offsetof(struct s1, i32) },
    { .tc = CDR_TYPECODE_ULONGLONG, .name = "u64",
      .offset = offsetof(struct s1, u64) },
    { .tc = CDR_TYPECODE_LONGLONG, .name = "i64",
      .offset = offsetof(struct s1, i64) },
    { .tc = CDR_TYPECODE_FLOAT, .name = "fl",
      .offset = offsetof(struct s1, fl) },
    { .tc = CDR_TYPECODE_DOUBLE, .name = "d",
      .offset = offsetof(struct s1, d) },
    { .tc = CDR_TYPECODE_CHAR, .name = "ch",
      .offset = offsetof(struct s1, ch) },
    { .tc = CDR_TYPECODE_BOOLEAN, .name = "b",
      .offset = offsetof(struct s1, b) },
    { .tc = CDR_TYPECODE_OCTET, .name = "o",
      .offset = offsetof(struct s1, o) },
    { .tc = CDR_TYPECODE_ENUM, .name = "anenumelement",
      .offset = offsetof(struct s1, anenumelement), .nelem = 3 },
    { .tc = CDR_TYPECODE_LONG, .name = "a", .label = a },
    { .tc = CDR_TYPECODE_LONG, .name = "b", .label = b },
    { .tc = CDR_TYPECODE_LONG, .name = "c", .label = c }
};
```

Arrays, sequences and strings

Following example contains some fixed, as well as dynamic array types and some dynamic and static strings.

The example in IDL looks like this:

```
const long MSGLEN = 20;

struct s2 {
    string<MSGLEN>  message;
    string          dynmsg;
    double          darr [10][2];
    sequence<float> floats;
};
```

This example uses a dynamic array, called a sequence. Allthough sequences are handled in more detail in the following chapter, a small introduction is given here in order to be able to understand the next code fragments.

Since there is no direct C-language representation of a dynamic array, a macro is used to define a sequence of float values.

```
#include "dds/dds_seq.h"
```

[4] This time we use the GCC extension for naming fields in constant declarations.

```
DDS_SEQUENCE (float, fseq);
```

This macro will expand to the following C-type definition:

```
typedef struct fseq_st {
    unsigned _maximum;
    unsigned _length;
    unsigned _esize;
    int      _own;
    float    *_buffer;
} fseq;
```

Sequences should not be handled directly. There is a large set of sequence handling prototypes defined in dds_seq.h that should be sufficient for the user and which allows almost direct use of all the sequence fields. See the next chapter for details.

The example uses this defined sequence:

```
#define MSGLEN 20

struct s2 {
    char message [MSGLEN];
    char *dynmsg;
    double darr [10][2];
    fseq floats;
};
```

This will result in the following TSMs:

```
#include <stddef.h>

static DDS_TypeSupport_meta tsm2[] = {
    { .tc = CDR_TYPECODE_STRUCT, .flags = TSMFLAG_DYNAMIC, .name = "s2",
      .size = sizeof (struct s2), .nelem = 4 },
    { .tc = CDR_TYPECODE_CSTRING, .name = "message", .nelem = MSGLEN,
      .offset = offsetof (struct s2, message) },
    { .tc = CDR_TYPECODE_CSTRING, .flags = TSMFLAG_DYNAMIC,
      .name = "dynmsg", .nelem = 0, },
      .offset = offsetof (struct s2, dynmsg ) },
    { .tc = CDR_TYPECODE_ARRAY, .name = "darr",
      .size = sizeof (double) * 20, .nelem = 10,
      .offset = offsetof (struct s2, darr)},
    { .tc = CDR_TYPECODE_ARRAY, .size = sizeof (double) * 2, .nelem = 2 },
    { .tc = CDR_TYPECODE_DOUBLE },
    { .tc = CDR_TYPECODE_SEQUENCE, .flags = TSMFLAG_DYNAMIC,
      .name = "floats", .nelem = 0,
      .offset = offsetof (struct s2, fseq)},
    { .tc = CDR_TYPECODE_FLOAT}
};
```

If a dynamic type, such as this, is given to the user via a DDS Reader, it is up to the user to determine where the dynamic data will be located. How this works will be described when sequences are discussed in more detail.

It is possible to constrain the size of embedded sequences in a TSM, if deemed necessary, by simply setting the *nelem* field to the maximum allowed number of elements in the sequence. If not constrained, the number of elements in the sequence may be unlimited, which is clearly impractical.

Union types

Union types in DDS are handled somewhat differently from normal C-unions, since DDS requires a special and unique discriminant value for each distinct union value. To make DDS union handling therefore somewhat easier, the DDS_UNION macro should be used:

```
DDS_UNION (union_type, discr_type, un);
```

Which expands to:

```
typedef struct un_st {
    discr_type discriminant;  /* Extra discriminant field. */
    union_type u;             /* Union data fields. */
} un;
```

The following example union will demonstrate how to handle TSMs for unions:

First we'll show the example using a union in IDL:

```
enum discr {
    ANINT,
    ASTRING
};

const long STRLEN = 20;

union dds_union switch (discr) {
    case ANINT: int i;
    case ASTRING: string<STRLEN> str;
};


struct u1s {
    dds_union aunion;
};
```

The same example, but resulting C-code:

```
#include "dds/dds_types.h"

typedef enum {
    ANINT,
    ASTRING
} discr;

#define STRLEN 20

typedef union {
    int  i;
    char str [STRLEN+1];
} u1;

DDS_UNION (u1, discr, dds_union);

typedef struct u1s_st {
    dds_union aunion;
} u1s;
```

Which will then result in the following TSMs:

```
#include <stddef.h>
```

```
static DDS_TypeSupport_meta u1s_tsm[] = {
    { .tc = CDR_TYPECODE_STRUCT, .name = "u1s",
      .size = sizeof (u1s), .nelem = 1 },
    { .tc = CDR_TYPECODE_UNION, .name = "aunion", .nelem = 2,
      .size = sizeof (dds_union), .offset = offsetof (u1s, aunion) },
    { .tc = CDR_TYPECODE_LONG, .name = "along",
      .offset = offsetof (dds_union, u), .label = ANINT },
    { .tc = CDR_TYPECODE_CSTRING, .name = "acstring",
      .offset = offsetof (dds_union, u),
      .size = STRLEN+1, .label = ASTRING }
};
```

Using Sequences

Introduction

Support for dynamic arrays, called sequences in DDS is supported directly via a set of sequence handling functions and macros. There is really no need for direct access to the sequence type itself, since the support functions should be able to provide all required accesses.

Sequences are strictly typed. Any previously defined type can be used as the element of a sequence, even sequences of sequences are allowed.

Whereas a number of programming languages have direct support for sequences, not all of them do. In C ++, sequences are handled using templates and overloading of standard operators. In C, there is no direct support in the programming language, and a macro is used to define the sequence and a set of access functions and macros is used for working with it.

Creating a new sequence type

Creating a new sequence type is done via the following macro:

```
#include "dds/dds_seq.h"

DDS_SEQUENCE(<element_type>, <sequence_type>);
```

Where the <element_type> can be whatever previously defined type.

The result will be the following type definition:

```
typedef struct name <sequence_type>_st {
    unsigned       _maximum;
    unsigned       _length;
    unsigned       _esize;
    int            _own;
    <element_type> *_buffer;
} <sequence_type>;
```

Although we will describe each sequence field in detail, it is not the intention that the user accesses these fields directly. This is just for understanding how sequences work. Actual handling of sequences should be via the access functions and macros, since this is an internal representation which might still be changed in future revisions of the software. Using the access functions and macros will thus ensure portability of user code to new Technicolor DDS versions.

The _maximum field specifies the maximum number of sequence elements that are currently available in _buffer. If the length of _buffer changes due to more elements being added to the sequence, the _maximum field will be adapted appropriately.

The _length field specifies the actual number of elements in the sequence. The _length field might vary between 0 and _maximum, but it may never be larger than _maximum. Resetting a sequence can be done simply by setting _length to 0. No memory needs to be allocated for adding elements unless the sequence contains (_length == _maximum) elements. In that case the _buffer field as well as _maximum will need to be updated before the element can be added.

The _esize field specifies the size of a sequence element. It is usually set when the instance is created, and initialized with *sizeof (<element_type>)*.

Although one might expect that the _esize type, which indicates the size of a particular sequence element, would be defined as a *size_t* type, this is not the case. This was a design choice. On 64-bit machines the compiler typically generates a 32-bit type for unsigned types, and a 64-bit type for *size_t* types, which we feel is clearly impractical in the context of DDS, where memory savings are more important, even on 64-bit machines. Besides, a 32-bit type is already capable of representing up to 4 Gigabytes of memory, and transfering chunks of more than 4 Gigabytes in a single message is clearly out of the question.

The _own field specifies ownership of the sequence element data. If _own is set, it means that the user of the sequence may change any element within the sequence. If not set, the sequence was passed to the called function by reference only, and the function is not supposed to change the sequence contents. See further for some specific use-cases.

The _buffer field is a pointer to an array of sequence elements. This array will initially be set to NULL, but will be allocated and reallocated appropriately when elements are subsequently appended to the sequence.

Creating and deleting sequences

Sequences can be either statically initialized when declared, or initialized explicitly. Static initialization is done using the DDS_SEQ_INITIALIZER() macro:

```
#include "dds/dds_seq.h"

DDS_SEQUENCE(float, FloatSeq);

static FloatSeq fs = DDS_SEQ_INITIALIZER(float);
```

The alternative explicit initialization uses the DDS_SEQ_INIT() macro:

```
#include "dds/dds_seq.h"

DDS_SEQUENCE(double, DoubleSeq);

int test(void)
{
    DoubleSeq dseq;

    DDS_SEQ_INIT(dseq);
    ...
}
```

Once a sequence has been used, i.e. elements were appended to it, it can be reset for reuse with the following generic function:

```
void dds_seq_reset(void *seq);
```

This function doesn't really cleanup all data, it just resets the sequence so that it becomes empty.

If a sequence is no longer needed, all its contained data can be disposed completely with the following generic function:

```
void dds_seq_cleanup(void *seq);
```

Both functions will only update the sequence descriptor. If the element type is a pointer type, the user should take care that all pointed to data is also disposed, possibly by walking over the sequence first, freeing the data that is pointed to, before calling the functions.

Don't use the *dds_seq_reset/cleanup()* functions on sequences that are created by Technicolor DDS. They are really meant for user-created sequences only. For DDS-originated sequences, although these functions might or might not work, depending on the element type, there are specific cleanup functions predefined that know exactly how to handle a correct cleanup. Therefore it is prudent to use only the DDS-specific cleanup functions in those situations, as described further.

Handling sequence elements

Once a sequence is initialized, data can be added to it, element by element in the following manner:

```
DDS_ReturnCode_t dds_seq_append(void *seq, void *value);
DDS_ReturnCode_t dds_seq_prepend(void *seq, void *value);
DDS_ReturnCode_t dds_seq_insert(void *seq, unsigned pos, void *value);
```

The first function adds an element at the end of the sequence, the second prepends an element, before all the existing elements, by shifting all these elements one position further in the sequence. The last function inserts a sequence element at a specified position in the sequence, hereby shifting all existing elements from that position onwards one position further.

A sequence can be completely initialized from an array of elements, sequence data can be stored in an array, using the following functions:

```
DDS_ReturnCode_t dds_seq_from_array(void *seq, void *array, unsigned len);
unsigned dds_seq_to_array(void *seq, void *array, unsigned len);
```

The first populates an empty sequence with array data. Note that the sequence must be empty for this to work properly. The second copies existing sequence data to an array, without changing the sequence contents.

Any existing element in a sequence can be replaced using the *dds_seq_replace()* function:

```
DDS_ReturnCode_t dds_seq_replace(void *seq, unsigned pos, void *value);
```

Removing elements from a sequence can be done with the following functions:

```
DDS_ReturnCode_t dds_seq_remove_first(void *seq, void *value);
DDS_ReturnCode_t dds_seq_remove_last(void *seq, void *value);
DDS_ReturnCode_t dds_seq_remove(void *seq, unsigned pos, void *value);
```

These functions remove elements from the sequence from the first, the last or from an arbitrary position in the sequence respectively.

Processing sequence elements often requires walking over the sequence, element by element. There are a number of ways to do this:

```
DDS_SEQ_FOREACH(seq,i) { ... }
```

```
DDS_SEQ_FOREACH_ENTRY(seq,i,ptr) { ... }
```

The first macro sets i initially at 0 and keeps incrementing it until the sequence end is reached. If the sequence is empty, nothing is done, otherwise the statement body will be called for each element in succession.

The difference between the two macros is that the latter sets ptr to the location of each element while walking over the sequence.

A more structured way of walking is with the following function:

```
void dds_seq_every(void *seq,
                   void (*func)(void *ptr, void *data),
                   void *data);
```

The *dds_seq_every()* function will call the subfunction func for each element, with the element pointer as the first argument and the passed data pointer as the second.

Although these functions are very generic, sometimes it is necessary to handle sequence elements in a more direct manner. Following macros give access to a sequence element at a specified position:

```
DDS_SEQ_ITEM(seq,i)
DDS_SEQ_ITEM_SET(seq,i,v)
DDS_SEQ_ITEM_PTR(seq,i)
```

The first simply returns the element of a sequence at position i. The second replaces the element at the given position with v.

Neither *DDS_SEQ_ITEM()* nor *DDS_SEQ_ITEM_SET()* validates the given element position. This must have been checked by the caller upfront.

The last macro returns a pointer to a sequence element location or NULL if the element is not in the sequence, so this does check the position.

Direct access to the sequence descriptor fields is possible using a number of macros, although one should take extreme care when using these.

```
DDS_SEQ_LENGTH(seq)
DDS_SEQ_MAXIMUM(seq)
DDS_SEQ_OWNED(seq)
DDS_SEQ_OWNED_SET(seq,o)
DDS_SEQ_ELEM_SIZE(seq)
```

The *DDS_SEQ_LENGTH()* returns the current sequence length, so this could be used to implement an alternative iterator. The *DDS_SEQ_MAXIMUM()* returns the current element buffer size. The *DDS_SEQ_OWNED()* and *DDS_SEQ_OWNED_SET()* are getter and setter functions for the sequence ownership field. *DDS_SEQ_ELEM_SIZE()* returns the size in bytes of a sequence element.

DDS-specific sequences

The former sequence operations are of course general manipulations of any sequence, user-defined or Technicolor DDS derived.

In practice, however, the user will also be confronted with the sequences that are defined in DDS itself. Technicolor DDS uses and has already defined a number of DDS sequences and sequence operations for specific DDS purposes.

Many of these sequences have a set of associated type-specific functions defined to make it easier for the user to create and dispose of them.

These functions come in a number of flavors and typically have the following prototypes:

```
<seq_type> *<seq_type>__alloc(void);
void <seq_type>__free(<seq_type> *arg);
void <seq_type>__init(<seq_type> *arg);
void <seq_type>__clear(<seq_type> *arg);
```

Here, the *<seq_type>__alloc()* function will attempt to allocate from the heap and reset appropriately a new sequence of the given type. If there is no more memory, a NULL value will be returned.

The *<seq_type>__free()* function is the reverse of the previous operation and will cleanly dispose the full sequence contents before releasing the sequence back to the heap.

The *<seq_type>__init()* function can be used to initialize a variable of the specific sequence type for the first time. It should not be called on an already initialized sequence since this will destroy (and leak) the contained data.

The last function, i.e. *<seq_type>__clear()* can be used to properly dispose of existing sequence data. It results in a clean empty sequence.

These functions are defined in Technicolor DDS for the following sequence types as well as for a number of container types that contain sequences:

Table 2: DDS Sequences and container types

Sequence or Container type	Element Type	Description
DDS_OctetSeq	unsigned char	Byte sequence, used as QoS parameter for User, Topic and Group data.
DDS_StringSeq	char *	A sequence of strings.
DDS_DataSeq	void *	An abstract data sequence.
DDS_SampleInfoSeq	DDS_SampleInfo *	Sequence of sample information as used in *_read*/*_take()* and *_return_loan()* functions.
DDS_InstanceHandleSeq	DDS_InstanceHandle_t	Sequence of instance handles.
DDS_DataReaderSeq	DDS_DataReader	Sequence of DataReaders.
DDS_GroupDataQosPolicy	-	Group data QoS contains a DDS_OctetSeq.
DDS_PartitionQosPolicy	-	Partition QoS contains a DDS_StringSeq.
DDS_TopicDataQosPolicy	-	Topic data QoS contains a DDS_OctetSeq.
DDS_UserDataQosPolicy	-	User data QoS contains a DDS_OctetSeq.
DDS_ParticipantBuiltinTopicData	-	Contains a DDS_UserDataQosPolicy element.
DDS_TopicBuiltinTopicData	-	Contains various dynamic data elements.
DDS_PublicationBuiltinTopicData	-	Contains various dynamic data elements.
DDS_SubscriptionBuiltinTopicData	-	Contains various dynamic data elements.
DDS_DomainParticipantQos	-	Contains a UserDataQosPolicy element.
DDS_TopicQos	-	Contains a TopicDataQosPolicy element.
DDS_SubscriberQos	-	Contains various dynamic data elements.

Sequence or Container type	Element Type	Description
DDS_PublisherQos	-	Contains various dynamic data elements.
DDS_DataReaderQos	-	Contains a DDS_UserDataQosPolicy element.
DDS_DataWriterQos	-	Contains a DDS_UserDataQosPolicy element.
DDS_ConditionSeq	DDS_Condition	Sequence of Conditions.

Using the _own field in DDS DataReader functions

In practice, the _own field is only used in the *DDS_DataReader_read/take()* functions to indicate to which layer the data buffers in the sequence belong. The following three mechanisms can be distinguished:

1. The user specifies own sample data buffers (_own = 1 in *DDS_DataReader_read/_take()* functions), and the dynamic contained data areas will be already allocated by the user, by setting the individual pointer fields to a non-NULL value. For the above example, the *dynmsg* and *dptr* fields and the floats._buffer would already be populated with pointers to valid and large enough data areas. The sequence maximum length (_maximum) must then also be sufficiently large to contain the maximum size of received elements. No *DDS_DataReader_return_loan()* is necessary after processing of the data samples and the user is completely responsible for managing its own data.
2. The user specifies, as before, sample data buffers (_own = 1 in *DDS_DataReader_read/_take()* functions), but the dynamic data fields, i.e. *dynmsg* and *dptr* fields and the floats._buffer are initialized to NULL. In this case, Technicolor DDS will allocate the dynamic memory using the builtin C-memory allocator (typically *malloc()*[5]). When done processing the samples, the user is responsible for cleaning up the data using the free memory function of the C-memory allocator (typically *free()*). As in the first case, no *DDS_DataReader_return_loan()* is necessary after processing of the data samples.
3. The user doesn't give sample data buffers (_own = 0 or both sample data and sample info sequences initialized to empty) in *DDS_DataReader_read/_take()* functions,. In this case Technicolor DDS is responsible for allocating all data, typically using another, more efficient but not disclosed, allocation strategy. Note that the use of this mechanism is a lot easier from the point of view of the caller, but it does mandate the use of *DDS_DataReader_return_loan()* when processing of data samples is done. It is also the more efficient method of processing received data since no copying is necessary - the user merely 'borrows' the buffers from DDS.

[5] In the future, the memory allocator function will be settable by the user, but this is not the case yet.

Using Dynamic types

Dynamic Shapes type.

The same Shapes type as defined before, can be constructed using the Extended types API [6] as demonstrated in the following code samples[7]:

```
#include "dds/dds_xtypes.h"

DDS_ReturnCode_t create_dyn_shape (void)
{
    DDS_TypeDescriptor       *desc;
    DDS_MemberDescriptor     *md;
    DDS_DynamicTypeMember    *dtm;
    DDS_AnnotationDescriptor *key_ad;
    DDS_DynamicTypeBuilder   sb, ssb;
    DDS_DynamicType          s, ss;

    desc = DDS_TypeDescriptor__alloc ();
    if (!desc)
        return (DDS_RETCODE_OUT_OF_RESOURCES);

    md = DDS_MemberDescriptor__alloc ();
    if (!md)
        return (DDS_RETCODE_OUT_OF_RESOURCES);

    key_ad = DDS_AnnotationDescriptor__alloc ();
    if (!key_ad)
        return (DDS_RETCODE_OUT_OF_RESOURCES);

    key_ad->type = DDS_DynamicTypeBuilderFactory_get_builtin_annotation ("Key");
    if (!key_ad->type)
        return (DDS_RETCODE_OUT_OF_RESOURCES);

    desc->kind = DDS_STRUCTURE_TYPE;
    desc->name = "ShapeType";
    sb = DDS_DynamicTypeBuilderFactory_create_type (desc);
    if (!sb)
        return (DDS_RETCODE_OUT_OF_RESOURCES);

    ssb = DDS_DynamicTypeBuilderFactory_create_string_type (128);
    if (!ssb)
        return (DDS_RETCODE_OUT_OF_RESOURCES);

    ss = DDS_DynamicTypeBuilder_build (ssb);
    if (!ss)
        return (DDS_RETCODE_OUT_OF_RESOURCES);

    md->name = "color";
    md->index = md->id = 0;
    md->type = ss;
    md->default_value = NULL;
    if (!md->type)
        return (DDS_RETCODE_OUT_OF_RESOURCES);
```

[6] The Extended types API is available when Technicolor DDS has been built with the XTYPES_USED build flag.

[7] In order not to clutter the code with error handling, the following examples don't properly cleanup all the resources in case errors should occur. Actual error handling when using dynamic types in production code should really be somewhat more rigidly.

```
    error = DDS_DynamicTypeBuilder_add_member (sb, md);
    if (error)
        return (error);

    dtm = DDS_DynamicTypeMember__alloc ();
    if (!dtm)
        return (DDS_RETCODE_OUT_OF_RESOURCES);

    error = DDS_DynamicTypeBuilder_get_member (sb, dtm, 0);
    if (error)
        return (error);

    error = DDS_DynamicTypeMember_apply_annotation (dtm, key_ad);
    if (error)
        return (error);

    free (dtm->descriptor.name);
    dtm->descriptor.name = NULL;
    DDS_DynamicTypeBuilderFactory_delete_type (dtm->descriptor.type);
    dtm->descriptor.type = NULL;
    md->name = "x";
    md->index = md->id = 1;
    md->type = DDS_DynamicTypeBuilderFactory_get_primitive_type
                                                 (DDS_INT_32_TYPE);
    if (!md->type)
        return (DDS_RETCODE_OUT_OF_RESOURCES);

    error = DDS_DynamicTypeBuilder_add_member (sb, md);
    if (error)
        return (error);

    md->name = "y";
    md->index = md->id = 2;
    md->type = DDS_DynamicTypeBuilderFactory_get_primitive_type
                                                 (DDS_INT_32_TYPE);
    if (!md->type)
        return (DDS_RETCODE_OUT_OF_RESOURCES);

    error = DDS_DynamicTypeBuilder_add_member (sb, md);
    if (error)
        return (error);

    md->name = "shapesize";
    md->index = md->id = 3;
    md->type = DDS_DynamicTypeBuilderFactory_get_primitive_type
                                                 (DDS_INT_32_TYPE);
    if (!md->type)
        return (DDS_RETCODE_OUT_OF_RESOURCES);

    error = DDS_DynamicTypeBuilder_add_member (sb, md);
    if (error)
        return (error);

    s = DDS_DynamicTypeBuilder_build (sb);
    if (!s)
        return (DDS_RETCODE_OUT_OF_RESOURCES);

    DDS_DynamicTypeBuilderFactory_delete_type (ssb);
    DDS_DynamicTypeBuilderFactory_delete_type (sb);
    dtype = s;
    shape_ts = DDS_DynamicTypeSupport_create_type_support (s);
    DDS_DynamicTypeBuilderFactory_delete_type (ss);
    DDS_TypeDescriptor__free (desc);
    DDS_MemberDescriptor__free (md);
    DDS_AnnotationDescriptor__free (key_ad);
    DDS_DynamicTypeMember__free (dtm);
    return (DDS_RETCODE_OK);
}
```

A Shapes dynamic data sample can then be constructed and written as shown in the following code fragment[8]:

```
    DDS_DynamicData dd;

    dd = DDS_DynamicDataFactory_create_data (dtype);
    if (!dd)
        fatal_printf ("Can't create dynamic data!");

    id = DDS_DynamicData_get_member_id_by_name (dd, "color");
    rc = DDS_DynamicData_set_string_value (dd, id, "RED");
    if (rc)
        fatal_printf ("Can't add data member(color)!");

    id = DDS_DynamicData_get_member_id_by_name (dd, "x");
    rc = DDS_DynamicData_set_int32_value (dd, id, 10);
    if (rc)
        fatal_printf ("Can't add data member(x)!");

    id = DDS_DynamicData_get_member_id_by_name (dd, "y");
    rc = DDS_DynamicData_set_int32_value (dd, id, 60);
    if (rc)
        fatal_printf ("Can't add data member(y)!");

    id = DDS_DynamicData_get_member_id_by_name (dd, "shapesize");
    rc = DDS_DynamicData_set_int32_value (dd, id, 25);
    if (rc)
        fatal_printf ("Can't add data member(shapesize)!");

    rc = DDS_DynamicDataWriter_write (w, dd, handle);
```

When a dynamic data is received, its member fields can be retrieved as shown in the following code:

```
    error = DDS_DynamicDataReader_take (dr, &sample, &info, 1, ss, vs, is);
    if (error) {
        if (error != DDS_RETCODE_NO_DATA)
            printf ("Unable to read: error = %s!\r\n", DDS_error (error));
        return;
    }
    if (DDS_SEQ_LENGTH (info)) {
        inf = DDS_SEQ_ITEM (info, 0);
        if (inf->valid_data && dyn_data) {
            dd = DDS_SEQ_ITEM (sample, 0);
            if (!dd)
                fatal_printf ("Empty dynamic sample!");

            id = DDS_DynamicData_get_member_id_by_name (dd, "color");
            error = DDS_DynamicData_get_string_value (dd, color, id);
            if (error)
                fatal_printf ("Can't get data member(color)!");

            id = DDS_DynamicData_get_member_id_by_name (dd, "x");
            error = DDS_DynamicData_get_int32_value (dd, &x, id);
            if (error)
                fatal_printf ("Can't get data member(x)!");

            id = DDS_DynamicData_get_member_id_by_name (dd, "y");
```

[8] The DDS_DynamicData_get_member_id_by_name() function calls are not really required since the id value is fixed when the type is created and is typically known in the program.

```
            error = DDS_DynamicData_get_int32_value (dd, &y, id);
            if (error)
                fatal_printf ("Can't get data member(y)!");

            id = DDS_DynamicData_get_member_id_by_name (dd, "shapesize");
            error = DDS_DynamicData_get_int32_value (dd, &shapesize, id);
            if (error)
                fatal_printf ("Can't get data member(shapesize)!");

            /* ... do something with the sample (color/x/y/shapesize) ... */
        }
    }
```

Dynamic sparse data type:

The sparse type with the following IDL specification:

```
//@Extensibility(MUTABLE_EXTENSIBILITY)
struct struct2m {
    uint16_t u16;   //@ID(10) //@Key
    int16_t i16;    //@ID(20)
    uint32_t u32;   //@Key
    int32_t i32;    //@ID(50)
    uint64_t u64;
    int64_t i64;
    float fl;
    double d;
    char ch;        //@ID(5) //@Key
};
```

can be constructed using the given C-code:

```
#include "dds/dds_xtypes.h"

#define ADD_FIELD(s,md,n,idx,i,t) md->name=n; md->index=idx; md->id=i;\
         md->type=DDS_DynamicTypeBuilderFactory_get_primitive_type(t);\
         fail_unless(md->type != NULL); \
         rc = DDS_DynamicTypeBuilder_add_member (s,md); \
         if (rc) return (rc)

DDS_ReturnCode_t set_key_annotation (DDS_DynamicTypeBuilder b,
                                     const char             *name)
{
    DDS_DynamicTypeMember           dtm;
    DDS_ReturnCode_t                ret;
    static DDS_AnnotationDescriptor ad = { NULL, };

    if (!b && ad.type) {
        DDS_AnnotationDescriptor__clear (&ad);
        return (DDS_RETCODE_OK);
    }
    DDS_DynamicTypeMember__init (&dtm);
    if (!ad.type) {
        ad.type = DDS_DynamicTypeBuilderFactory_get_builtin_annotation ("Key");
        if (ad.type == NULL)
            return (DDS_RETCODE_OUT_OF_RESOURCES);
    }
    ret = DDS_DynamicTypeBuilder_get_member_by_name (b, &dtm, name);
    if (ret)
        return (ret);

    ret = DDS_DynamicTypeMember_apply_annotation (&dtm, &ad);
    if (ret)
        return (ret);
    free (dtm.descriptor.name);
    dtm.descriptor.name = NULL;
    DDS_DynamicTypeBuilderFactory_delete_type (dtm.descriptor.type);
    dtm.descriptor.type = NULL;
    return (DDS_RETCODE_OK);
}

DDS_ReturnCode_t set_id_annotation (DDS_DynamicTypeBuilder b,
                                    const char             *name,
                                    DDS_MemberId           id)
{
    DDS_DynamicTypeMember           dtm;
    DDS_ReturnCode_t                ret;
    unsigned                        n;
    char                            buf [12];
    static DDS_AnnotationDescriptor ad = { NULL, };

    if (!b && ad.type) {
        DDS_AnnotationDescriptor__clear (&ad);
        return (DDS_RETCODE_OK);
    }
    DDS_DynamicTypeMember__init (&dtm);
    if (!ad.type) {
        ad.type = DDS_DynamicTypeBuilderFactory_get_builtin_annotation ("ID");
        if (!ad.type)
            return (DDS_RETCODE_OUT_OF_RESOURCES);
    }
    n = snprintf (buf, sizeof (buf), "%u", id);
    if (!n)
        return (DDS_RETCODE_OUT_OF_RESOURCES);

    ret = DDS_AnnotationDescriptor_set_value (&ad, "value", buf);
    if (ret)
        return (ret);

    ret = DDS_DynamicTypeBuilder_get_member_by_name (b, &dtm, name);
    if (ret)
        return (ret);

    ret = DDS_DynamicTypeMember_apply_annotation (&dtm, &ad);
    if (ret)
        return (ret);

    free (dtm.descriptor.name);
    dtm.descriptor.name = NULL;
    DDS_DynamicTypeBuilderFactory_delete_type (dtm.descriptor.type);
    dtm.descriptor.type = NULL;
    return (DDS_RETCODE_OK);
}

DDS_ReturnCode_t set_ext_annotation (DDS_DynamicTypeBuilder b,
                                     const char             *ext)
{
    DDS_ReturnCode_t                ret;
    static DDS_AnnotationDescriptor ad = { NULL, };

    if (!b && ad.type) {
        DDS_AnnotationDescriptor__clear (&ad);
        return (DDS_RETCODE_OK);
    }
    if (!ad.type) {
        ad.type = DDS_DynamicTypeBuilderFactory_get_builtin_annotation
                                                              ("Extensibility");
        if (!ad.type)
            return (DDS_RETCODE_OUT_OF_RESOURCES);
    }
    ret = DDS_AnnotationDescriptor_set_value (&ad, "value", ext);
    if (ret)
        return (ret);

    ret = DDS_DynamicTypeBuilder_apply_annotation (b, &ad);
    return (ret);
}

DDS_ReturnCode_t create_dyn_struct2m (void)
{
    DDS_TypeSupport        *ts;
    DDS_DynamicTypeBuilder sb2;
    DDS_DynamicType        s2;
    DDS_TypeDescriptor     *desc;
    DDS_MemberDescriptor   *md;
    DDS_ReturnCode_t       rc;

    /* 1. Create the type. */
    desc = DDS_TypeDescriptor__alloc ();
    if (!desc)
        return (DDS_RETCODE_OUT_OF_RESOURCES);

    desc->kind = DDS_STRUCTURE_TYPE;
    desc->name = "dstruct2m";

    sb2 = DDS_DynamicTypeBuilderFactory_create_type (desc);
    if (!sb2)
        return (DDS_RETCODE_OUT_OF_RESOURCES);

    md = DDS_MemberDescriptor__alloc ();
    if (!md)
        return (DDS_RETCODE_OUT_OF_RESOURCES);

    ADD_FIELD (sb2, md, "i16", 0, 99, DDS_INT_16_TYPE);
    rc = set_id_annotation (sb2, "i16", 20);
    if (rc) return (rc);

    ADD_FIELD (sb2, md, "u32", 1, 21, DDS_UINT_32_TYPE);
    rc = set_key_annotation (sb2, "u32");
    if (rc) return (rc);

    ADD_FIELD (sb2, md, "i32", 2, 100, DDS_INT_32_TYPE);
    rc = set_id_annotation (sb2, "i32", 50);
    if (rc) return (rc);

    ADD_FIELD (sb2, md, "u16", 0, DDS_MEMBER_ID_INVALID, DDS_UINT_16_TYPE);
    rc = set_id_annotation (sb2, "u16", 10);
    if (rc) return (rc);

    rc = set_key_annotation (sb2, "u16");
    if (rc) return (rc);

    ADD_FIELD (sb2, md, "u64", 5, 51, DDS_UINT_64_TYPE);
    ADD_FIELD (sb2, md, "i64", 5, DDS_MEMBER_ID_INVALID, DDS_INT_64_TYPE);
    ADD_FIELD (sb2, md, "fl",  6, 53, DDS_FLOAT_32_TYPE);
    ADD_FIELD (sb2, md, "d",   7, 54, DDS_FLOAT_64_TYPE);
    ADD_FIELD (sb2, md, "ch",  8, 55, DDS_CHAR_8_TYPE);
    rc = set_id_annotation (sb2, "ch", 5);
    if (rc) return (rc);

    rc = set_ext_annotation (sb2, "MUTABLE_EXTENSIBILITY");
    if (rc) return (rc);

    s2 = DDS_DynamicTypeBuilder_build (sb2);
    if (!s2) return (DDS_RETCODE_OUT_OF_RESOURCES);
    DDS_DynamicTypeBuilderFactory_delete_type (sb2);

    ts = DDS_DynamicTypeSupport_create_type_support (s2);
    if (!ts) return (DDS_RETCODE_OUT_OF_RESOURCES);

    DDS_TypeDescriptor__free (desc);
    DDS_MemberDescriptor__free (md);
    return (DDS_RETCODE_OK):
}
```

A data sample of this type can then be constructed as follows:

```
#define SET_FIELD(dd,id,type,v) rc=DDS_DynamicData_set_##type##_value (dd,id,v);\
         if (rc) return (rc)

DDS_ReturnCode_t write_sample(DDS_DynamicType s2,
                              DDS_DataWriter  w,
                              DDS_HANDLE      h)
{
    DDS_DynamicData        dd;
    DDS_ReturnCode_t       rc;

    dd = DDS_DynamicDataFactory_create_data (s2);
    if (!dd)
        return (DDS_RETCODE_OUT_OF_RESOURCES;

    SET_FIELD (dd, 50, int32, -5);
    SET_FIELD (dd, 10, uint16, 0xDEAD);
    SET_FIELD (dd, 20, int16, INT16_MIN);
    SET_FIELD (dd, 21, uint32, UINT32_MAX);
    SET_FIELD (dd, 52, int64, 100);
    SET_FIELD (dd, 53, float32, 0.5f);
    SET_FIELD (dd, 54, float64, 100e-5);
    SET_FIELD (dd, 5, char8, 'd');
    SET_FIELD (dd, 51, uint64, 5010000);

    rc = DDS_DynamicDataWriter_write (w, dd, h);
    return (rc);
}
```

5 WaitSets and Listeners

Topics:

- *Listener functions*
- *WaitSets*

DDS allows two distinct methods in order to be notified of events, and both can be used simultaneously, i.e. Listener functions and WaitSets.

There are advantages and disadvantages to each of these two mechanisms, and care must be taken in order to properly handle additional DDS function calls in either case.

Following discussion will explain each mechanism in detail.

Listener functions

The first and foremost mechanism is the Listener function. This mimics a bit the Unix *signal* mechanism, in that it results in an asynchronous callback function in a thread context that is not the thread of the user's application, so it is very important that some precautions are taken to ensure proper use of this mechanism.

Listener callbacks can be specified on different entities as well as on encompassing entities on top of the former. For a listener to be active, both the function callback must be set and the Entity event mask indicating the specific listener must be set. The primary entities and their optional listeners are listed below:

Table 3: Listener functions

Entity	Specific Listener functions
DataReader	*requested_deadline_missed(), requested_incompatible_qos(), sample_rejected(), liveliness_changed(), data_available(), subscription_match(), sample_lost()*
Subscriber	*data_on_readers()*
DataWriter	*offered_deadline_missed(), offered_incompatible_qos(), liveliness_lost(), publication_match()*
Topic	*inconsistent_topic()*

The entities encompassing these primary entities inherit the listener functions of the contained entities. For example, the Subscriber entity has both its own specific listener (*data_on_readers()*), but also inherits every DataReader listener functions. The same can be set of the Publisher entity, since it also contains the listener functions of the DataWriter.

The DomainParticipant then inherits all listener functions, since it is a parent of all the Topics, DataReaders, DataWriters as well as of all Publishers and Subscribers in the domain.

When multiple Listeners are available on all levels, the general rule is that the most specific Listener will always be called. The listeners on the higher levels are only called if there's no listener on a more specific entity. There is however a single exception of this rule for the Subscriber.

The *data_on_readers()* listener is somewhat special, because even though there might be more specific *data_available()* listeners on the contained DataReaders, if this specific Subscriber listener is enabled, it will be called first. The reason for this is that it is still possible to call the specific *data_available()* listeners from within the *data_on_readers()* callback function, using the *DDS_Subscriber_notify_datareaders()* function. Refer to the DCPS specification for more details on this.

As stated before, care must be taken as to which functions are used within a listener callback. Since these callbacks occur from within a Technicolor DDS internal thread, it is very important that a callback function tries to do its work as fast as possible and doesn't go to sleep for a long time. If there is a lot of work to do, or if the workload might block for a long time it is necessary to dispatch this work to another thread, or just use the alternative WaitSet mechanism which was really meant for these use-cases.

Technicolor DDS ensures that the state of DDS is valid for a number of function calls. However, function calls that would block will return immediately with an error code, and function calls that delete entities may not be called.

Within these restrictions, it is possible, and this is effectively one of the more commonly used use-cases for listeners, to read data directly from a DataReader cache from within the listener callback, and even to send data on another DataWriter. For this, it is sufficient to call any of the *DDS_DataReader_read/take()* function calls, followed by an optional *DDS_DataReader_return_loan()* when finished with it.

WaitSets

WaitSets are a lot easier to work with and are a lot safer than listeners. In fact, a WaitSet can be seen as a kind of 'overloaded' *select()* or *poll()* function call. Both time-outs as well as waiting for data and waiting for other events, including notifications from other threads, can be handled properly using this mechanism.

WaitSets don't have any of the restrictions that are specific to listener callbacks. Everything can be done, i.e. every DDS function may be called, since everything occurs in the thread context of the application thread.

Handling WaitSets requires a bit more preparation work, however. Before a WaitSet can be used, it needs to be created properly, and this requires a number of function calls.

The typical workflow is as follows:

- Call *DDS_WaitSet__alloc()* to create a new WaitSet.
- Call *DDS_ConditionSeq__alloc()* or use *DDS_ConditionSeq__init()* on an already existing variable to create/initialize an empty Condition sequence. This is needed in order to add conditions that we are interested in to the WaitSet conditions.
- Add one or more conditions using the *DDS_WaitSet_attach_condition()* function. Conditions come in many flavors, such as:
  1. ReadConditions. These are DataReader conditions, where we specify a DataReader and the set of View, Instance and Sample states that we are interested in.
  2. QueryConditions. A superset of a ReadCondition, where in addition to the states, it is possible to specify an SQL-based expression on the data samples that we're interested in. DDS will only signal that the condition is ready when both the expression matches and the requested states are available.
  3. StatusConditions. Any state can be signalled as a condition on any entity, similar to the various Listener callbacks:

Table 4: Status bits

Status	Description	Entity
INCONSISTENT_TOPIC_STATUS	Topic is not compatible with an existing topic, or with a discovered topic with the same type.	Topic
OFFERED_DEADLINE_MISSED_STATUS	When Deadline QoS is enabled, Writer has missed a deadline for writing instance data.	Writer
REQUESTED_DEADLINE_MISSED_STATUS	Idem, where Reader missed a deadline for reading instance data.	Reader
OFFERED_INCOMPATIBLE_QOS_STATU	Offered Writer QoS settings are incompatible with Reader's QoS.	Writer

Status	Description	Entity
REQUESTED_INCOMPATIBLE_QOS_STATUS	Reader QoS settings are incompatible with Writer QoS.	Reader
SAMPLE_LOST_STATUS	Sample was lost due to a slow/late reader.	Reader
SAMPLE_REJECTED_STATUS	Sample was not accepted due to resource limits.	Reader
DATA_ON_READERS_STATUS	One of the DataReaders has data available.	Subscriber
DATA_AVAILABLE_STATUS	Data is available for reading.	Reader
LIVELINESS_LOST_STATUS	When Liveliness QoS is enabled, Writer was too slow with sending data.	Writer
LIVELINESS_CHANGED_STATUS	idem, where Reader is not responsive anymore.	Reader
PUBLICATION_MATCHED_STATUS	Writer has found a matching Reader.	Writer
SUBSCRIPTION_MATCHED_STATUS	Reader has found a matching Writer.	Reader

4. GuardConditions. A guardcondition can be used to wakeup a thread that is waiting for a number of Conditions in the WaitSet. An example use would be waiting for both one or more sockets being ready as well as on a number of DDS Conditions. The main DDS waiting thread would be blocked on the WaitSet, with an additional I/O monitoring thread waiting on *select()*, *read()* or *fread()* functions. When data is available via the I/O thread, this data/event could be appended to an input queue, followed by a *set_trigger_value()* function call to unblock the main DDS waiting thread.

- Call *DDS_WaitSet_wait ()* with a suitable time-out. This function will return only if either of the given Conditions became active, or a time-out occurred. It is easy to see which Conditions were ready, since the list of ready Conditions is returned in a sequence of Conditions which can be checked individually, based on the type of condition. When processing is done, the *DDS_WaitSet_wait()* can be called again, for as many times as needed, until the program finishes.

6 Technicolor DDS memory handling

Topics:

- *Memory pools*
- *Configuring memory pools*

The way in which Technicolor DDS uses its data internally can be configured extensively in order optimize it for specific application components.

The following description will attempt to explain the mechanism that is used and will describe how to adapt DDS for the applications.

Memory pools

In order to speed up regular allocations and deallocations of memory blocks, a generic acceleration mechanism was introduced in Technicolor DDS.

For each type of often used memory, a pool is defined that is able to group a collection of preallocated memory chunks, chained together as a linked list of identical sized blocks.

Whenever a memory block is requested from a pool, the first available block is returned and the linked list contains one less element.

If a memory block is requested, but the pool has been drained completely, it is configurable as to what will happen then.

- If the pool limit was set to exactly the number of preallocated memory blocks for that pool, a NULL pointer is returned and an internal out-of-memory statistics counter is incremented.
- If the limit is higher than the number of preallocated memory blocks, a second list (the grow list) is checked for spare memory blocks. If there are memory blocks in that list, the new block will be taken from the grow list.
- If the grow list is empty and the memory constraints allow it, a new block will be allocated via a standard memory allocation function and that one will be returned.

When a memory block is released back to a pool from which it was allocated, it is either:

- Put back in front of the preallocated block linked list, if it was effectively a preallocated block, or
- it was allocated and the grow list is not full yet, it is added to the grow list, or
- it is released via a standard memory release function.

It is fully configurable by the user as to:

- How many memory blocks are preallocated in a pool.
- What the absolute limit of blocks is that may be in use for a pool.
- The maximum size of the grow list in relation to the preallocated list size (as a percentage, i.e. 0..infinity).

A number of statistics are kept per pool, as shown in the following table:

Table 5: Memory Pool statistics

Name	Description
MPUse	Maximum number of blocks that were ever allocated from the preallocated block list.
CPUse	Number of preallocated blocks that are currently in use (i.e. allocated).
MXUse	Maximum number of extra blocks that ever needed to be allocated in addition to the preallocated blocks.
CXUse	Number of extra blocks that are currently in use.
Alloc	Total number of extra block allocations that were done.
NoMem	Total number of times that a block was requested, but the pool was empty.

The names in the table correspond with the names that are displayed in the output of the ′`spool`′ or ′`spoola`′ debug commands.

If the -DFORCE_MALLOC compile option is given when Technicolor DDS is built, no preallocated blocks will be allocated and all allocations/frees will use the standard memory allocater.

Configuring memory pools

The memory pools can be configured from application components using the following Technicolor DDS functions (located in *<dds/dds_aux.h>*:

```
DDS_ReturnCode_t DDS_get_default_pool_constraints (DDS_PoolConstraints *pars,
                                                   unsigned            max_factor,
                                                   unsigned

 grow_factor);
```

This function will initialize the pars parameter, which is a structure containing all pool parameters, to suitable values in accordance with the maximum and grow factors.

The max_factor specifies the relation between the total number of allocatable blocks and the number of preallocated pool blocks as an extra percentage factor. I.e. if max_factor is set to 0, the maximum number of blocks will be equal to the number of preallocated blocks. If max_factor is set to 100, the maximum number of blocks will be twice the number of preallocated blocks. If set to ~0, the maximum number of blocks will not be limited and all available system memory may be used.

The grow_factor specifies the maximum length of the grow list for each of the memory pools as a percentage of the number of extra items that must not be returned.

```
DDS_ReturnCode_t DDS_set_pool_constraints (const DDS_PoolConstraints *pars);
```

This function should be used to apply the pool constraint parameters.

These functions should be used before the first DDS DomainParticipant is created, or they will not have any effect.

The following table summarizes the pool parameters that can be modified:

Table 6: Memory Pool Parameters

Name	Minimum	Maximum	Description
Domain	*<na>*	max_domains	Local DDS DomainParticipant
Subscriber	min_subscribers	max_subscribers	Local DDS Subscriber
Publisher	min_publishers	max_publishers	Local DDS Publisher
LocalReader	min_local_readers	max_local_readers	Local DDS DataReader
LocalWriter	min_local_writers	max_local_writers	Local DDS DataWriter
Topic	min_topics	max_topics	DDS Topic
FilteredTopic	min_filtered_topics	max_filtered_topics	DDS Filtered Topic

Name	Minimum	Maximum	Description
TopicType	min_topic_types	max_topic_types	DDS Topic Type.
ReaderProxy	min_reader_proxies	max_reader_proxies	Remote DDS DataReader Proxy
WriterProxy	min_writer_proxies	max_writer_proxies	Remote DDS DataWriter Proxy
RemoteParticipant	min_remote_participants	max_remote_participants	Discovered DDS DomainParticipant
RemoteReader	min_remote_readers	max_remote_readers	Discovered DDS DataReader
RemoteWriter	min_remote_writers	max_remote_writers	Discovered DDS DataWriter
PoolData	min_pool_data	max_pool_data	Data Buffers for sample storage
RxBuffer	*<na>*	max_rx_buffers	Receive buffers for RTPS
Change	min_changes	max_changes	Cache Change descriptor.
Instance	min_instances	max_instances	Cache Instance descriptor.
ApplicationSample	min_application_sample	max_application_sample	Application sample references.
LocalMatch	min_local_match	max_local_match	Local matching DataReader for a DataWriter.
CacheWaiter	min_cache_waiters	max_cache_waiters	Maximum threads waiting for a Cache.
CacheTransfer	min_cache_transfers	max_cache_transfers	Samples being queued between two local Data History Caches.
TimeFilter	min_time_filters	max_time_filters	Active Time Filter.
TimeInstance	min_time_instances	max_time_instances	Time Filter Instance.
String	min_strings	max_strings	Unique string descriptor
StringData	min_string_data	max_string_data	Unique string data area
Locator	min_locators	max_locators	Locator descriptor (see RTPS).
QoS	min_qos	max_qos	Unique DDS Quality of Service data.
List	min_lists	max_lists	Dynamic list
ListNode	min_list_nodes	max_list_nodes	Dynamic list node.
Timer	min_timers	max_timers	Timer
IPv4Socket	*<na>*	max_ipv4_sockets	IPv4 socket
IPv4Address	*<na>*	max_ipv4_addresses	IPv4 address
WaitSet	min_waitsets	max_waitsets	DDS WaitSet
StatusCondition	min_statusconditions	max_statusconditions	DDS StatusCondition
ReadCondition	min_readconditions	max_readconditions	DDS ReadCondition
QueryCondition	min_queryconditions	max_queryconditions	DDS QueryCondition
GuardCondition	min_guardconditions	max_guardconditions	DDS GuardCondition

Name	Minimum	Maximum	Description
Notification	min_notifications	max_notifications	Outstanding Listener notification.
TopicWait	min_topicwaits	max_topicwaits	Threads waiting for a Topic
Guard	min_guards	max_guards	Used if any of the Liveliness, Deadline, Lifespan and AutoPurge QoS parameters are active.
DynamicType	min_dyn_types	max_dyn_types	A DDS Dynamic type (see X-Types)
DynamicData	min_dyn_samples	max_dyn_samples	A DDS Dynamic sample (see X-Types)

To make it a bit more easy for the user to configure these pool parameters, the Technicolor DDS debug shell can be used when an application has been running in steady-state for some time. The resulting `spool` (or `spoola`) debug command output can then be used to figure out how much memory was actually used in DDS itself by looking at the MPUse and MXUse parameters (see before). The total memory ever used will then be the sum of these two parameters.

7 Environment variables

Topics:

- *Environment variables influencing Multicast behavior*
- *Environment variables that determine usable IP interfaces*
- *Environment variables influencing the scope of IP addresses*
- *Environment variables influencing IPv4 and IPv6 modes of operation.*

Environment variables influencing Multicast behavior

Table 7: Multicast environment variables

Environment Variable	Description
TDDS_IP_MCAST_TTL	Can be used to change the default Time-To-Live value for multicast datagrams. If not specified, this TTL value is set to 1. Possible values: 1..255.
TDDS_IP_MCAST_DEST	Specifies the IP address of the interface that must be used in order to route Multicast IP packets. This variable must be set in cases when there is no default gateway defined and there is no valid Multicast route present in the system. If set, it strictly overrides any automatic Multicast forwarding mechanisms that are present in the system, and can thus be used to restrict DDS access to other interfaces. It is allowed to specify the loopback address (e.g. 127.0.0.1) as a crude way of preventing Technicolor DDS from reaching other nodes outside the own node.
TDDS_IP_NO_MCAST	This environment variable can be set to indicate which DDS peers do not accept Multicast traffic or on which networks it is not wise to use Multicast traffic (such as wireless networks). The argument can be a combination of IP address and IP network specifications similar to the specification that is used in TDDS_IP_ADDRESS and TDDS_IP_NETWORK (see further).
TDDS_IPV6_MCAST_HOPS	Specifies the maximum number of hops for outgoing IPv6 multicast packets. It should be set to a value between 0 and 255.
TDDS_IPV6_MCAST_INTF	This option can be used to set the default outgoing interface for IPv6 multicast packets. Contrary to the TDDS_IP_MCAST_DEST environment variable, the argument should be a valid interface name (such as lo or eth0).
TDDS_IPV6_GROUP	This variable should be used in case the default IPv6 Discovery multicast address (FF03::80) is not available. It should be set as a normal IPv6 address specification.

Examples:

export TDDS_IP_MCAST_TTL=2

Sets the Time-To-Live value for IPv4 Multicast datagrams to 2, so Multicast IP packets will be able to be routed 1 node further from the own node, but not further.

export TDDS_IP_MCAST_DEST=10.0.0.64

Set the Multicast route to the IP interface on which the address 10.0.0.64 is present, disregarding default IP Multicast routes. Note that this interface must be present for Multicast routing to be possible.

export TDDS_IP_NO_MCAST=10.0.0.32+31;144.11.0.12

Technicolor DDS will not send Multicast datagrams to DDS peers in the range 10.0.0.32..10.0.0.63, nor to a DDS peer located on 144.11.0.12.

The mechanism that Technicolor DDS uses to send data to multiple peers depends on this environment variable. If a datagram must be sent to a group of DDS peers with at least one peer in the given range, Technicolor DDS will choose to use specific Unicast packets to each individual destination within the group, even to those that do allow Multicast. If none of the destinations is mentioned in the variable, multicast will still be used.

Environment variables that determine usable IP interfaces

The general behavior of Technicolor DDS as to the use of IP interface addresses is very simple. On startup, all active IP interfaces are queried, and if a valid IP address is found, this will be used. If no valid IP address is found, Technicolor DDS will print out a warning message ("No IPv4 addresses available - waiting for address assignment!") and wait until it finds a usable IP address. [9][10]

In practice, this means that all IP addresses on a device will be used, even some that are really not meant to be used (such as *vmWare* addresses when a *vmWare* player is operating in bridged mode). This makes Technicolor DDS extremely open to the IP-network for all DDS domains, which might not always be a good idea from a security point of view.

To alleviate these issues somewhat, there are two ways to cope with this.

Firstly, it is possible to configure Technicolor DDS at build time to always filter away *vmWare* addresses. Since this is only useful in environments where we know that the *vmWare* addresses are effectively an issue, and is even wrong when *vmWare* is used in other modes (routing with or without NAT), this is rather crude and somewhat of a hack.

Another, second, mechanism was therefore introduced which is a lot more powerful and which allows the user to specify allowed addresses with a high degree of freedom.

This advanced mechanism is based on two environment variables: TDDS_IP_ADDRESS and TDDS_IP_NETWORK, which together allow the user to create address and network blacklists as well as whitelists.

[9] The assignment of IP-addresses is currently done on startup of Technicolor DDS only. This means that if addresses change (due to a DHCP lease being lost or due to some explicit configuration changes), connectivity will be lost. This is a known issue and will need to be fixed somewhere in the future by monitoring IP address changes and by automatically coping with changes when they occur.

[10] Only IPv4 addresses can be specified currently with these environment variables. Since public IPv6 addresses are not assigned by default to an IPv6 connection, and since Link-Local IPv6 addresses (which are generated automatically at startup) are not picked up by Technicolor DDS (see TDDS_IPV6_SCOPE), this is not seen as a problem. When there is a clearly perceived need for IPv6 address selection, support for this might still be added by introducing new environment variables.

Table 8: IP addresses environment variable

Environment variable	Description
TDDS_IP_ADDRESS	This variable can be used to specify IP addresses which may (whitelist) or may not (blacklist) be used. Multiple addresses can be specified, and addresses can even differ per DDS domain.

This is a filter specification which is applied to the found IP interface addresses in the system. It can not be used to override existing network addresses.

The EBNF syntax of the TDDS_IP_ADDRESS environment variable is as follows:

```
<tdds_addr_env> = "TDDS_IP_ADDRESS=" <address_spec> {';' <address_spec> }
<address_spec> = [<not>] [<domain_id> ':'] <ipv4addr>
<not> = '~' | '^'
<domain_id> = <number>                          -- 0..230
<ipv4addr> = <ipv4octet> { '.' <ipv4octet> }     -- 2..4 IPv4 octets
<ipv4octet> = <number>                          -- 0..255
```

If less than 4 octets are specified, 0's will be inserted between the last octet and the octet before that one in order to get to a length of 4 octets. Example: 10.5 is converted to 10.0.0.5.

Examples:

export TDDS_IP_ADDRESS=10.100;120.10.6.90

Only addresses 10.0.0.100 and 120.10.6.90 may be used by Technicolor DDS.

export TDDS_IP_ADDRESS=100:10.1.0.200

For domain 100, only address 10.1.0.200 may be used. Other domains can use any address.

export TDDS_IP_ADDRESS=^10:10.5;10.5

For domain 10, all addresses except 10.0.0.5 may be used. Other domains may only use 10.0.0.5.

Table 9: IP networks environment variables

Environment variable	Description
TDDS_IP_NETWORK	This variable can be used to define IP networks which may or may not be used. Similar to TDDS_IP_ADDRESS, multiple specifications may be given, and the specification may differ per DDS domain.

The EBNF syntax of the TDDS_IP_NETWORK environment variable is as follows:

```
<tdds_net_env> = TDDS_IP_NETWORK=<network_spec> {';' <network_spec> }
<network_spec> = [<not>] [<domain_id> ':'] <ipv4net>
<ipv4net> = <ipv4addr> |                     -- A single IPv4 address
               <ipv4addr> '/' <width> |      -- A masked IPv4 subnet.
               <ipv4addr> '+' <count>        -- Range of IPv4 addresses.
<width> = <number>                           -- 4..31 mask bits.
<count> = <number>                           -- 1..64K-1 extra addresses.
```

Contrary to IPv4 address specifications, specifying less than 4 octets in a network address will result in 0-byte padding after the last octet to get a length of 4 octets. Example: 10.5 is converted to 10.5.0.0.

Examples:

export TDDS_IP_NETWORK=10/8

Only addresses in the range 10.0.0.0 to 10.255.255.255 may be used.

export TDDS_IP_NETWORK=10/8;~10.1/16;50:100.10/14;10.1.1.20+24

All addresses in the range 10.0.0.0 to 10.255.255.255 that are not in the range 10.1.0.0 to 10.1.255.255, with the exception of 10.1.1.20 to 10.1.1.44 may be used for all domains. In addition, domain 50 may use addresses in the range 100.10.0.0 to 100.13.255.255.

Both TDDS_IP_ADDRESS and TDDS_IP_NETWORK can be used simultaneously, giving the opportunity to the user to specify both addresses and network ranges.

If none of these environment variables is specified, all valid interface addresses that are detected will be used.

The existence of address and/or network specification environment variables implies that filtering is used on all the detected interface IP addresses. This filtering uses a matching algorithm that operates in the following manner:

- If no match is found with any address or network specifications, the address will not be used.
- If a single match occurs, the address or network specification will either accept the address or reject it, depending on the presence of the <not> operator. If <not> was given in the specification, the address is rejected, otherwise it is accepted.
- If multiple matches occur, the first valid match with the longest prefix mask will be used.

Both blacklisting (using the <not> operator) and whitelisting (listing usable addresses/networks) can be used simultaneously as described above.

Environment variables influencing the scope of IP addresses

Table 10: Address Scope Environment variables

Environment Variable	Description
TDDS_IP_SCOPE	May be used to limit the scope of IPv4 addresses that will be used as source addresses and that will be announced to peer participants. If not specified, all source addresses with Link, Site, Organisation and Global scope will be used (see further). A valid scope expression should be specified as explained below.
TDDS_IPV6_SCOPE	Specifies the scope of IPv6 addresses that will be used as source addresses and that will be announced to peer participants. If not specified all source addresses with Site, Organisation and Global scope will be used. A valid scope expression should be used, as explained below.

A scope expression has the form of: *<ScopeNumber>*-*<ScopeNumber>*, where *<ScopeNumber>* is a decimal number from 0 to 4, with the following significance of each number:

Table 11: Address scope numbers

Number	Scope	Description
0	Node	Node-local addresses are addresses that should never be used outside of a a node as a source address (the loopback address).
1	Link	Link-local addresses are typically generated by a system. They should not be used outside the link on which they were defined. On IPv4: auto-ip addresses. On IPv6: the automatically generated link addresses.
2	Site	Site-local addresses should not be used outside a site. For IPv4, this includes the 192.168/16 and the 10/24 address ranges. For IPv6, the address scope is defined with the address.
3	Organisation	Organisation-local addresses should not be used outside a specific organisation. This is typically not known for IPv4 addresses, but can be specify with IPv6 addresses.
4	Global	Global or Public addresses can always be reached from anywhere over the Internet.

The scope expression thus makes it easy to restrict the scope of IPv4 or IPv6 addresses that may be used by Technicolor DDS.

The expression only influences which local IPv4 (or IPv6) addresses will be announced to DDS peers and does not mean that addresses learned from other peers that are outside this scope will be ignored when received from peer DDS participants. It is not clear currently whether a mechanism for ignoring these remote addresses would be useful.

Environment variables influencing IPv4 and IPv6 modes of operation.

Table 12: IPv4 and IPv6 mode environment variables

Environment Variable	Description
TDDS_IP_MODE	Specifies the default mode of operation of IPv4. By default, IPv4 operation is enabled and preferred over IPv6. The argument should be set to one of the valid IP modes as explained below.
TDDS_IPV6_MODE	Specifies the default mode of operation of IPv6. By default, if IPv6 is enabled at compile time, it is enabled, but not preferred. The argument should be set to one of the valid IP modes.

Valid arguments for TDDS_IP_MODE and TDDS_IPV6_MODE are:

Table 13: IPv4 and IPv6 mode arguments

Mode	Description
DISABLE[D]	The specific IP transport is disabled and remains passive. IP addresses of the transport type will not be used and discovery of this transport will be inactive.
ENABLE[D]	The specific IP transport is enabled, but is not prioritized. This means that the transport will function and will announce its source addresses, but it will only be used if the other transport has no means of reaching a certain destination.
PREFER[RED]	The specific IP transport is enabled and preferred.

If both IPv4 and IPv6 have priority, then IPv4 will still be used.

Although described in uppercase in the mode table, alternatively all lower-case modes can be used as well.

The combination of mode settings will, in the current implementation, enable only one discovery protocol. If IPv6 is enabled in cooperation with IPv4, discovery will be done over IPv4, even though IPv6 might be preferred. Only if IPv4 is disabled will Discovery be performed over IPv6. The rationale here is that Discovery over IPv4 is clearly defined in the DDS specification as the default Discovery protocol.

8 Technicolor DDS components

Topics:

- *Overview*
- *Central Discovery Daemon*
- *Shapes demo*
- *Bandwith program*
- *Latency program*
- *IDL to TSM converter*
- *The DCPS functionality test program*
- *The DCPS API test suite*
- *Library build options*

Technicolor DDS is delivered as a single build component, consisting of a dynamically loadable shared library containing all core DDS functionality, with a number of prebuilt utility and demo programs on the supported platforms.

There is usually no need for changing the component. However, there are still some compile/link-time options available for ´tweaking´ the behavior somewhat. It is even possible to link statically if there would be a need for it.

Overview

Shared library

The `libdds.so.0` library is a dynamically loadable shared library that can be built on Unix/Linux platforms to provide the core Technicolor DDS functionality. Applications link to the library via a *-ldds* linker options.

Central Discovery Daemon

The Central Discovery Daemon (`cdd`) is responsible for keeping per-node Discovery data central.

This prevents local components to all keep their own copy of Discovery data which would multiply the amount of memory needed for storing Discovery data with the number of components. The amount of memory used can be very high in cases where a large number of distinct topics are used.

Shapes demo

The shapes program (`shapes`) demonstrates DDS interoperability between different DDS vendors.

It is a technology demo, showing in real-time various shapes (Circle, Square and Triangle) that are moving around. In the program, the user can create new shapes, as well as subscribe to shapes.

A bandwidth test program

The bandwidth test program (`bw`) can be used to test DDS performance.

The program can be started either as a data producer or as a data consumer, or even both, so it can test performance between multiple DDS nodes and processes. It attempts to stress the system by sending bursts of data at controlled speeds, while verifying that data is successfully received at the remote end.

Latency test program

This latency test program (`latency`) is able to accurately measure the latency between two DDS components.

It sends a number of data samples that are looped by a remote entity and registers the time it took for arrival of the response. When the test is completed, a report is generated on the measured latency.

IDL to TSM conversion utility

An IDL parser is available that is very useful for generation of TSM data. Manual conversion of IDL specifications are typically very error-prone. A tool that does this automatically is therefore extremely useful.

The DCPS functionality test program

This program, called `dcps`, is used internally to test out various functionalities of DDS between different application components. At compile time, lots of options are available for creating Readers and Writers with various kinds op QoS settings. As many instances of the program as necessary can be run simultaneously. It can be interacted with in a dynamic manner in order to start and stop data transfers from individual DataWriters and to monitor received data from individual DataReaders.

DCPS API test suite program

The API test program (`api`) tests out the full DDS DCPS API in order to verify that the Technicolor DDS implementation works correctly. It is a limited conformance test suite, but still manages to test all public DCPS API functions at least once. It is limited since it doesn't have full coverage of all DDS code, but runs very quickly and is used as one of the tests in the battery of tests that are performed when a new DDS build is done.

Central Discovery Daemon

DDS Discovery mechanism

DDS uses the Simple Discovery protocol (*SPDP/SEDP*) to propagate Participant, Topic and Endpoint (i.e. Reader/Writer) information to all other discovered DDS peers. This is needed, since a DDS-component can only 'match' with peer endpoints if this information is locally available, i.e. discovered.

Although this is perfectly ok if there are only a limited amount of DDS components in a device or if there is plenty of memory in a device when there are a lot of DDS components, this can become a real problem on low-memory footprint devices when a lot of topics need to be discovered.

The amount of memory that is used on each device for discovery data (e.g. meta-data) would in practice be: the number of components times the total number of topics and endpoints in the DDS domain times the memory needed to store each of the endpoints.

In Technicolor DDS, this memory usage is not very high, i.e. less than 80 bytes per discovered endpoint, which is a lot less than what other DDS'es require. However, even this small amount can quickly add up! Assuming 30 components and 2000 endpoints in a domain on a single device, for example, this would end up occupying: 30 x 2000 x 80 = 4.8MB for discovery data alone.

In practice, a component has typically no interest in the majority of the discovered data. Only the data that matches its own endpoints is really needed for correct DDS operation. For small components that have only a few matching endpoints, having to store all the unused discovery data can be a real burden.

A naive approach to filter discovered information might be to just discard everything that is of no interest yet, but this simply doesn't work. The discovered endpoint/topic info might be needed afterwards. This can happen if the order of endpoint creation is different between the component and its peers and the component is a bit slower (intentionally or not) than its peers. A match would then not happen and communication would not be possible.

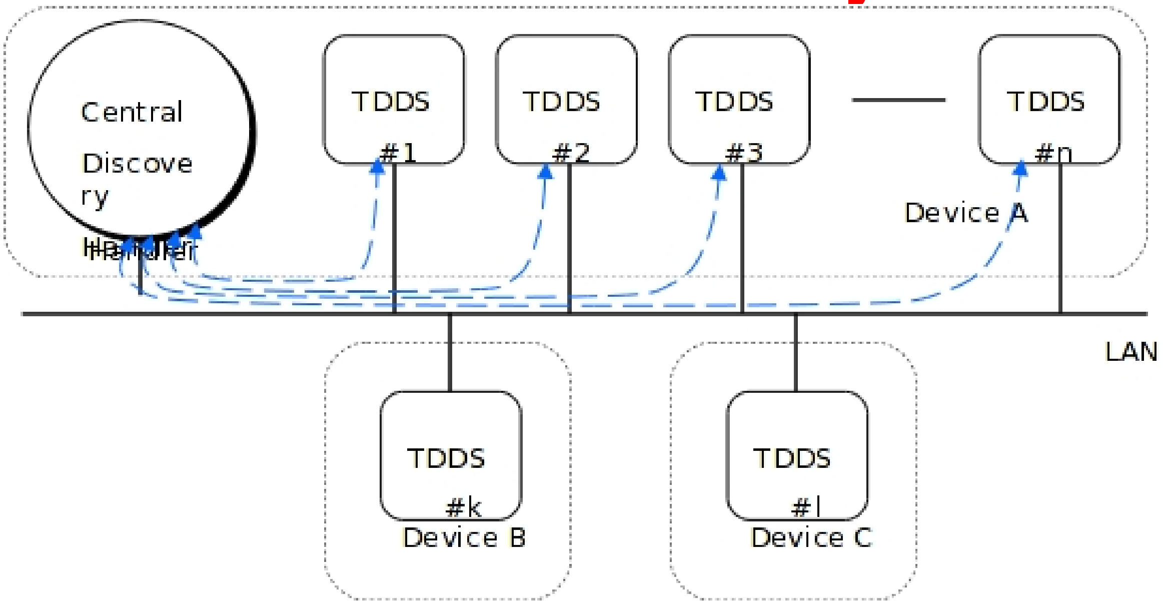
In fact, most embedded DDS's just disable the Simple Discovery protocol and use static (manually configured) discovery data. Although this can work with a limited amount of Topics/Endpoints, overall it is very cumbersome and also very error-prone, and it doesn't scale at all.

As shall be seen further, the idea of the Central Discovery handler solves the memory usage problems, while still allowing a fully dynamic Discovery mechanism.

Centralizing Discovery data

By centralizing the DDS Discovery meta-data on a device, a lot of memory can be saved. Using the numbers presented in the previous chapter and assuming that all Discovery data is needed only twice (not an unrealistic assumption), once in the Central Discovery handler and once distributed over the components of a device, the following numbers come up: 2 x 2000 x 80 = 320 Kbytes, i.e. we can save almost 4.5 Mbytes of memory!

See below for an overview of the Central Discovery Daemon in action:



Of course, it is not all positive news since there are also a few disadvantages associated with a ′central′ Discovery data store:

- It introduces a central point of failure in the device, which goes against the DDS ′philosophy′. If something happens with the central discovery daemon (**CDD**), crashes or otherwise, no more new matches will happen, although existing matches will continue to work.
- It slows down the time before discovery happens in the components, since there is more communication overhead and the matching decisions for late joining components are taken by the **CDD** instead of by the component itself.
- Some security issues might pop up if remote participants filter on participant USER_DATA to determine whether communication is allowed. The **CDD** is visible remotely as a separate participant and might be filtered out. Also, it might send matching information to local components, even though this was not allowed. Although the design of the daemon partially alleviates some of these issues, some problems might still occur. Some further study might be necessary to look at the various security issues.

CDD design principles

The following design principles are used:

- All communication between device components and the **CDD** will be over extra DDS builtin Central Discovery Endpoints, similar to the SPDP and SEDP endpoints.
- The Central Discovery endpoints will be invisible to external devices/components, since like builtin Discovery Endpoints they are not published.
- If a component starts up without a **CDD** being present (the **CDD** registers itself in the Technicolor DDS shared memory), or if specifically instructed not to use an existing **CDD** (via either an environment variable, or via a command-line flag), it will not use Centralized Discovery and just behave as a normal DDS component, storing all Discovery data.
- The **CDD** itself *will* be visible to external devices/components, but it will never announce Subscriptions or Publications over the normal SPDP/SEDP builtin endpoints.
- Since it is visible to the external world as an extra participant, the **CDD** captures all discovered information coming from components on the same device as well as from components on other devices, which it stores into its Discovery Data store.

- If the **CDD** detects that other participants on the *same* device publish new, i.e. not yet received Publications/Subscriptions, an additional action will happen; i.e. the Data store will be checked for new matches with existing endpoints in other components (local or remote). If a match is then found, this will be notified on one of the builtin Central Discovery endpoints.
- If the **CDD** detects that other participants on a *different* device publish new information, the **CDD** will not send anything to local components, since these still have their normal Discovery builtin endpoints and will react autonomously on the received information.

Starting `cdd`

The program, when used on Unix/Linux operating systems, daemonizes itself by default, and detaches itself from the standard I/O channels. It also then logs its operation to the Syslog service, and can be stopped by sending a Hangup (HUP) signal to it. This daemonizing behavior can be prevented with the `-D` option.

The program accepts a number of options that can be shown by typing `cdd -h`.

The following options can be given:

Table 14: `cdd` program options

Option	Description
`-i <domain>`	Specifies the DDS Domain Identifier for which the Central Discovery function must be enabled. If not specified, Domain Identifier 0 is used.
`-f`	Continue if the Daemon is already running. This option is present to recover in cases where a previous instance of `cdd` stopped unexpectedly without proper cleanup. In case this happens, the shared memory region would wrongly indicate an active `cdd`, preventing the daemon from starting again.
`-r`	Reset the shared memory region before starting up. This can be used when DDS application components didn´t complete properly, thereby messing up the count of active in-node DDS participants.
`-d`	Display/dump all Discovery data whenever changes are detected. Since this can lead to a lot of data being dumped, it is not advised to do this when daemonized (see `-D` option).
`-D`	Don´t daemonize on startup, but behave as a normal user program. This option could be used while debugging, since cdd then displays all output on the normal terminal output channel, and its debug shell (when enabled) can be used directly from the command line.
`-p <name>`	Specifies the name of a pipe on which `cdd` will send its *pid* when fully activated. This can be used by an init process to know the *pid* in order to be able to close `cdd`, but also to block until `cdd` is completely initialized before starting other DDS-based application processes.
`-v`	Verbose operation flag. This causes the program to log the main actions while it´s active.
`-vv`	Very verbose operation flag. If enabled, cdd wil give a lot of information while running, probably a lot more than desired. Don´t use this while daemonized.
`-h`	Displays some information on program usage.

Shapes demo

Introduction

The shapes demo works by creating per basic shape form a DDS DataWriter, having a different instance per shape color (color = key), and periodically moving the shape in a specific direction (by publishing new shape data with different X and Y coordinates) until a wall is hit, whereafter the shape reverses direction.

Since only a limited number of shapes are defined, there are only up to 3 shape writers:

- square
- circle
- triangle

Subscribing to a shape form is done by creating a DDS DataReader, which then monitors the shape movements and draws the shape on the screen. There are also only three shape readers due to the limited number of shape forms.

The user is able to create new shapes dynamically via a builtin text GUI with context-specific menus.

It is also possible to create shape readers and shape writers from the command line at program startup.

A builtin debug shell can be enabled to monitor the DDS dynamic data usage, such as DataWriter and DataReader cache contents as well as other data while the program is active.

The shapes program can be started in as many terminals and on as many nodes as required. All shape writers will be able to communicate with every shape reader, whether locally, or over the network.

Starting `shapes`.

The `shapes` program can be run either using a full-screen text GUI interface, or in batch mode, where it just displays updates as messages on standard output.

The program accepts a number of options as shown by typing `shapes -h`.

The following options can be given:

Table 15: `shapes` program options

Option	Description
`-p <shape <color>`	Publish a shape (square, circle or triangle) with the given color (red, green, yellow, blue, magenta, cyan).
`-s <shape>`	Subscribe to a shape (square, circle or triangle).
`-t`	Trace messages, i.e. start in batch mode without a GUI.
`-b`	Black-on-white display instead of the default white-on-black.
`-x <strength>`	Use exclusive ownership QoS with the given strength value.
`-d`	Start in debug mode.
`-v`	Verbose operation flag: logs overall functionality.
`-vv`	Very verbose flag: logs detailed functionality.
`-h`	Displays some information on program usage.

GUI mode

In GUI mode, the program starts up with the following screen:

```
Shapes demo (c) 2011, Technicolor   Publish  Subscribe  Command  Quit
                                  #####################################
Subscriptions:                    #                                   #
                                  #                                   #
                                  #                                   #
Publications:                     #                                   #
                                  #                                   #
                                  #                                   #
                                  #                                   #
                                  #                                   #
                                  #                                   #
                                  #                                   #
                                  #                                   #
                                  #                                   #
                                  #                                   #
                                  #                                   #
                                  #                                   #
                                  #                                   #
                                  #                                   #
                                  #                                   #
                                  #                                   #
                                  #####################################
```

A limited number of commands are accepted here, as single characters:

When 'p' is typed, the Publish menu is displayed which gives the option to create a new Shapes instance writer. All local shape instance writers are listed under the Publications header of the GUI screen.

If 's' is typed, the Subscribe menu is displayed which allows to create a new Shapes reader, with the option of filtering received shapes based on shape coordinates. All local shape readers are listed under the Subscriptions: header of the GUI screen.

If 'c' is typed, the shapes program enters the DDS Debug Shell, and displays the shell prompt. The user can then display DDS Debug commands and examine the debug responses. Published shapes are still updated, and received shapes are still monitored, so this allows an in depth examination of a running DDS application.

If 'q' is typed, the program exits immediately.

Publish menu

The Publish menu gives the user the opportunity to add new shape publications. It will query the shape type (Square, Circle or Triangle) and the shape color, and allows the user to select both parameters with a single keystroke from a menu of possible options.

It is entered via the 'p' command from the main GUI screen.

Following example creates a red square:

```
Shapes demo (c) 2011, Technicolor


    Publish

    Shape type:

        Square
        Circle
        Triangle

        or <esc> to return
```

```
         Choice?
```

After 's' is typed, the following menu is presented:

```
Shapes demo (c) 2011, Technicolor


   Publish

   Shape type = Square
   Shape color:

       Red
       Green
       Yellow
       Blue
       Magenta
       Cyan

       or <esc> to return

         Choice?
```

After 'r' is typed a red square is added to the list of local publications, together with the current X and Y coordinates, which will constantly be updated while the program is running.

No square shape will be visible on the map if there is no shapes reader for square shapes. This needs to be created using the Subscribe menu.

Subscribe menu

The Subscribe menu allows the user to create new Shape readers, optionally specifying a coordinate-based shape data filter.

It is entered via the 's' command in the main GUI screen.

Following example creates a Square shapes reader:

```
Shapes demo (c) 2011, Technicolor


   Subscribe

   Shape type:

       Square
       Circle
       Triangle

       or <esc> to return

         Choice?
```

After 's' is typed, the following is shown:

```
Shapes demo (c) 2011, Technicolor


   Subscribe

   Shape type = Square
   Filter:

       Yes
       No
       or <esc> to return

         Choice?
```

If the user types 'n', a non-filtered Shapes reader is created which is shown on the top GUI screen.

If 'y' is typed, the coordinates of the filter are requested. First the X-coordinate of the left corner of the visible square, next the X-coordinate of the right corner of the visible square, then the Y-coordinate of the top corner, followed by the Y-coordinate of the bottom corner:

```
Shapes demo (c) 2011, Technicolor


   Subscribe

   Shape type = Square
   Filter = Yes
   X1 (5..238) : 10
   X2 (2..240) : 120
   Y1 (5..254) : 50
   Y2 (4..256) : 200
```

After the final coordinate is specified (end each coordinate input with <enter>), the filtered reader is created and the top GUI screen shows the visual shape rectangle:

```
Shapes demo (c) 2011, Technicolor   Publish  Subscribe  Command  Quit
                                  #####################################
Subscriptions:                    #                                   #
                                  #                                   #
   Square                         #                                   #
                                  # ## # # # # # # ##                 #
Publications:                     # #               #                 #
                                  #                                   #
   Square   RED     115 202       # #               #                 #
                                  #                                   #
                                  # #               #                 #
                                  #                                   #
                                  # #               #                 #
                                  #                                   #
                                  # #               #                 #
                                  #                                   #
                                  # ## # # # # # # S#                 #
                                  #                                   #
                                  #                                   #
                                  #                                   #
                                  #                                   #
                                  #####################################
```

As can be seen from the screen display, whenever a reader is created and there are active shape writers, whether they are locally defined or generated by other DDS shapes writers, these shapes will be displayed on the top GUI screen in the map area.

Shapes are displayed in the map as a colored character, denoting the form of the shape ('S', 'C' or 'T'). The color of the square will correspond with the defined square color.

Command mode

The shapes command mode shows the builtin Debug shell as shown below:

```
Shapes command shell -- type 'h' or '?' for help.
>
```

Here, both the ′quit′ command, as well as an additional ′menu′ command is available to go back to the main GUI screen.

Refer to the Debug shell chapter for more details on how to use the command shell.

Bandwith program

Introduction

The DDS bandwitdh test program (`bw`) is used to measure the bandwidth between two or more DDS components, where one is a sender of data and the others are receivers of the data.

It can be used to send timed data bursts at the Writer side in order to optimize the bandwidth to reach the highest throughput for a predetermined time period.

A large number of test parameters can be tweaked:

- Data is sent either reliably or in a best-effort manner.
- The size of the data chunks can be selected up to 32KBytes per data item.
- The number of data chunks in each burst is selectable.
- The delay between successive data bursts is selectable.
- The total delay of a test can be specified.

Test methodology

The same bandwidth program can be used either as a data producer or as a data consumer, or both, depending on the startup options given when the program is started.

It doesn′t matter which of the two is started first, since the producer waits till all consumers are present before starting with the data transfers.

Separate command and status topics are used to:

1. The data producer commands the data consumers that a test is initiated or stopped.
2. The producer retrieves status information from the data consumers as to whether samples are missed.

Once all consumers are ready, the producer sends data of the specified size in bursts of a given length. When a burst is finished, the producer delays for some configured time, and while the test is not done yet, it repeats the burst.

When the test is finished, test results are displayed.

Starting the `bw` program

The bw program only runs from the command line and accepts a number of program options that can be shown by typing `bw -h`:

Table 16: `bw` program options

Option	Description
`-c`	Act as a data consumer.
`-p`	Act as a data producer.

Option	Description
`-cp|pc`	Act as both, i.e. producer and consumer are in the same process, and exchange information locally. This is the default mode of operation if no options are given.
`-n <nrdrs>`	Number of data consumers listening for data. Default = 1.
`-t <time>`	Total test time in seconds. Default test time = 5 seconds.
`-s <size>`	Size of each data chunk in bytes. Default size = 1 byte. Sizes can be specified from 1 up to 32768 bytes.
`-b <count>`	Maximum number of data samples in each burst. Default burst size = 16. Burst size may be set from 1 up to 1000 data samples.
`-d <usec>`	Maximum delay in #seconds between each successive data burst. Default delay = 100#s.
`-r`	If given, data transfers will be reliable, i.e. data will be retransmitted when lost.
`-k <history>`	Number of samples to keep in the history. Default is all samples.
`-g`	Display a histogram of received samples in time.
`-f`	Repeat the tests forever.
`-v`	Verbose operation - logs overall functionality.
`-vv`	Very verbose - logs detailed functionality.
`-h`	Display some information on program usage.

Latency program

Introduction

The DDS latency test program (`latency`) is used to measure the latency, i.e. the end-to-end time delay that is consumed before samples are actually delivered from a data producer to a data consumer over DDS.

Test methodology

Testing latency is done using three distinct test entities:

1. A *Controller*, which controls the next two entities.
2. A *Looper*, which just loops back data to whoever sends data to it.
3. A *Generator*, which sends the samples to the looper entity.

The controller entity waits until all entities are present before starting a new test by sending a *START* command to the Generator.

The generator, on receiving this command, starts sending data samples periodically to the looper. When the looper receives a sample, it simply returns it.

When all samples are sent and looped, the generator stops and a latency report is generated.

All test entities can be combined in one program run by specifying appropriate options. Typical run examples are:

- `latency -cgl -s 1024` would run all functionalities in one program instance locally and test latency for a 1K sample size.
- `latency -cg -n 100` on one node and `latency -l` on another node would test latency between two nodes, with the controller and generator combined in the first program, sending 100 samples.

Starting the `latency` program

The latency program only runs from the command line and accepts a number of program options that can be shown by typing `latency -h`:

Table 17: `latency` program options

Option	Description
`-c`	Controller mode (controls the other entities).
`-g`	Generator mode (sends data samples periodically and measures the latency from the returned, i.e. looped data sample.
`-l`	Looper mode (default setting if no mode is specified). In this mode, the program simply loops samples back to the generator.
`-n <nsamples>`	Number of samples to sent for latency calculation. If not specified, 50 samples are sent.
`-s <size>`	Data sample size in bytes. Default size = 1. Data sizes can range from 1 up to and including 32768.
`-d <msec>`	Delay between each successive sample in milliseconds (default = 100ms).
`-r`	Use reliable transfers instead of best-effort operation.
`-k <history>`	Number of samples to keep in the history. Default history is to keep all samples.
`-i`	Dump latency information for each sample. If not specified, only summary data containing minimum, maximum, average and median latency information is given.
`-v`	Verbose operation. Logs overall functionality.
`-vv`	Very verbose operation, logging detailed functionality.
`-h`	Display some information on program usage.

IDL to TSM converter

Introduction

The TSM type that is used in Technicolor DDS was not really intended to be hand-written by users, although it is certainly possible to do so. It should really be seen as an intermediate type representation that is typically generated using special-purpose tools.

The complete core middleware development framework allows users to specify application components from an XML-definition file. This file contains sufficient information on the data types that will be used by applications. TSMs are then typically generated from this XML file using special core middleware tools.

In order to allow proper testing of the Technicolor DDS typecode framework, a need for a standalone TSM generation tool was identified.

The choice of a source type definition language fell on IDL, since it is a standardized data type mechanism that is source language independent, and using this would allow Technicolor DDS to take advantage of already existing IDL-based type definitions. An additional pleasant side-effect, of course, is that it is now possible to use Technicolor DDS as a standalone DDS middleware as well.

Using the `idlparser` program

The IDL parser program expects an IDL input, which is specified with the filename argument. If not specified, the parser will read from standard input.

If no output file is specified (using `-o`), the parser assumes *out.c* as the default output file, which will be automatically created or overwritten if it already exists.

The parser accepts a number of program options that can be seen by typing `idlparser -h`:

Table 18: `idlparser` program options

Option	Description
`-o <outfile>`	Specifies the output file. If not specified, *out.c* is assumed.
`-t`	Generate test code for testing the generated type.
`-h`	Display program usage and options.

Example usage

Suppose the following input is used in *shapes.idl*:

```
struct ShapeType {
   string<128> color; //@Key
   long x;
   long y;
   long shapesize;
};
```

Then giving the command: `idlparser -o shapes.c shapes.idl`

Would result in the following *shapes.c* file contents:

```
#include <stdlib.h>
#include <stdio.h>
#include <dds/dds_dcps.h>

typedef struct _ShapeType_st {
   char color[128];
   int x;
   int y;
   int shapesize;
} ShapeType;

static DDS_TypeSupport_meta ShapeType_tsm [] = {
{CDR_TYPECODE_STRUCT   , TSMFLAG_KEY, "ShapeType", sizeof (struct _ShapeType_st),
 0, 4, 0, NULL},
{CDR_TYPECODE_CSTRING  , TSMFLAG_KEY, "color", 128, offsetof (struct
 _ShapeType_st, color), 0, 0, NULL},
{CDR_TYPECODE_LONG     , 0, "x", 0, offsetof (struct _ShapeType_st, x), 0, 0,
 NULL},
{CDR_TYPECODE_LONG     , 0, "y", 0, offsetof (struct _ShapeType_st, y), 0, 0,
 NULL},
{CDR_TYPECODE_LONG     , 0, "shapesize", 0, offsetof (struct _ShapeType_st,
 shapesize), 0, 0, NULL}
};
```

The DCPS functionality test program

Introduction

The dcps test program (`dds`) is used for testing of the Technicolor DDS middleware.

It allows to configure various different QoS combinations and allows the use of configurable data transfers between multiple DataReaders and DataWriters, locally, as well as on multiple hosts.

Program usage

By default, the program will behave as a single DataReader, waiting for data to arrive.

If started with the `-w` option, the program will create both a DataReader and a DataWriter, and unless the `-p` option is given, the program will start sending data immediately at a rate of 1 data sample per second.

Depending upon the `-v` option, sent and received data samples will be displayed on standard output.

Since the DDS Debug Shell is started by default, the user can examine DDS entities in detail while the test is running.

Some extra shell commands are present to allow some program-specific interactions:

- The 'pause' command allows to stop sending data samples on a DataWriter.
- The 'resume [<nsamples>]' command allows to continue sending data samples on the DataWriter, either continously or for a given number of samples.
- The 'delay <n>' command updates the delay between the generation of successive data samples. Whereas this is by default set to 1 second, it can be set to values of 1 up to <maxint> milliseconds.
- The 'asp <d>' command can be used to invoke a DDS_DomainParticipant_assert_liveliness() function on the given domain.
- The 'ase <endpoint>' command can be used to invoke a DDS_DataWriter_assert_liveliness() function on the given endpoint handle.
- The 'quit' command can be used to exit the program, while properly cleaning up all resources.

Command line options

The program accepts a number of options as can be shown by typing `dds -h`.

The following options can be specified:

Table 19: `dds` program options

Option	Description
`-w`	Act as DataWriter, sending data samples.
`-r`	Act as a DataReader. This is the default, so doesn't need to be specified.
`-0..3`	Number of local DataReaders in writer mode. Default is 1.
`-s <size>`	Size of data samples sent in writer mode. By default, a 4-byte counter is sent, followed by string.

Option	Description
`-m <size>`	Maximum data size of data samples. If set, the data sample sizes will increase from the -s argument and this one, and restart again.
`-n <count>`	Maximum number of data samples to generate/receive. When the given amount is reached, the program stops sending/receiving or quits, depending on the -q option.
`-q`	Quit when all packets are sent (writer side) or received (reader side).
`-f`	Flood mode when given. In flood mode, the writer doesn't wait, but sends as fast as possible.
`-d <msec>`	Maximum delay to wait between sending samples in milliseconds. Default delay is 1 second (1000.ms). Minimum delay is 10ms.
`-p`	For a DataWriter: start in paused state (i.e. wait until a 'resume' command is entered.
`-t`	Trace transmitted/received messages.
`-l`	Disable lower-layer RTPS functionality, so it can be used on the build farm as one of the tests in the automatic test suite.
`-v`	Verbose operation, logging overall functionality.
`-vv`	Very verbose operation, i.e. detailed functionality will be logged.
`-h`	Displays program usage.

The DCPS API test suite

Introduction

The DCPS API test suite (api) attempts to test all public DDS DCPS API functions with a number of different parameters.

It is intended to be run as a batch test, testing the total DCPS functionality, topic by topic. However, it is also possible to indicate specific topics in order to test only those specific functionalities.

Test methodology

Simple tests, if possible, just call functions within the to be tested topic.

More complex tests typically requires prologue and epilogue code to setup the edge conditions to be able to call a specific API function. Sometimes this requires quite an elaborate setup consisting of DomainParticipants, Topics, Publishers, Subscribers, DataReaders and DataWriters.

For the more complex tests, such as the discovery and data tests, up to 3 different domain participants within the same domain need to be created.

The advantage of this method is that everything runs in a single test program, while still enabling fundamental DDS functions such as RTPS messaging and state machines as well as active DDS Discovery mechanisms.

Starting the test suite

As mentioned above, the API test suite is divided in a number of individual test topics. These test topics can be shown with the `api -l` command, which then lists all the available topics:

Table 20: API test suite topics.

Topic	Description
`extra`	Auxiliary functions, e.g. Technicolor DDS-specific extensions to the DDS DCPS API.
`seq`	Tests for the Technicolor DDS-specific sequence functions.
`type`	Type registration tests.
`factory`	DomainParticipantFactory tests.
`participant`	DomainParticipant tests.
`topic`	Topic tests.
`filter`	Content-filtered topic tests.
`multitopic`	Multitopic tests[11].
`pub`	Publisher tests.
`sub`	Subscriber tests.
`writer`	DataWriter tests.
`reader`	DataReader tests.
`data`	Complex Data exchange tests.

If the program is started with as arguments a number of these topics, only the given topics will be tested.

The test suite accepts a number of startup options, that can be displayed with `api -h`.

Following options can be specified:

Table 21: API test suite options.

Option	Description
`-l`	Lists all test topics.
`-d <msec>`	Maximum delay to wait for responses (10..10000 milliseconds).
`-n <num>`	Total number of times that the tests need to run.
`-r`	Disable the lower-layer RTPS functionality.
`-s`	Enable the DDS Debug Shell.
`-v`	Verbose operation, logging overall functionality.
`-vv`	Very verbose, i.e. tries to log detailed test funtionality.
`-h`	Display program usage.

[11] Since Technicolor DDS doesn't support Multitopics yet, this just tests that the correct error codes are returned.

Library build options

Following build options are available directly from the build system:

- DEBUG=1 This option automatically enables the debug shell as well as the debug server.
- <TBC - Bruno to fill in some other options if available?>

9 DDS Debug shell

Topics:

Introduction

The DDS debug shell can be activated in a number of ways and allows users to examine/verify the internal workings of the Technicolor DDS middleware component.

It can be started, either directly from a specific application program linked to the Technicolor DDS middleware, or remotely via the DDS Debug server over a Telnet session. Both ways to connect to the shell result in approximately the same command interface with only minor differences.

Command Overview

The Debug shell, once activated will show a prompt and will wait for debug commands.

For example, when remotely connected over telnet, the following will be displayed:

```
Welcome to the TDDS Debug shell.
Type 'help' for more information on available commands.
>
```

Typing the 'help' command will show the list of DDS Debug commands that are supported in an active shell:

```
TDDS Debug shell -- (c) Technicolor, 2012
Following commands are available:
    stimer                Display the timers.
    sstr                  Display the string cache.
    spool                 Display the pools.
    spoola                Display the pools (extended).
    scx                   Display connections.
    sloc                  Display locators.
    sdomain <d> <lf> <rf> Display domain (d) info.
                          <lf> and <rf> are bitmaps for local/remote info.
                          1=Locator, 2=Builtin, 4=Endp, 8=Type, 10=Topic.
    sdisc                 Display discovery info.
    sdisca                Display all discovery info (sdisc + endpoints)
    sqos                  Display QoS parameters.
    stopic <d> [<name>]   Display Topic information.
    sendpoints            Display the DCPS/RTPS Readers/Writers.
    scache <ep>           Display an RTPS Endpoint Cache.
    sdcache <ep>          Display a DCPS Endpoint Cache.
    qcache <ep> <query>   Query cache data of the specified endpoint:
                          where: <ep>: endpoint, <query>: SQL Query string.
    sproxy <ep>           Display the Proxy contexts of an entry.
    seqos <ep>            Display endpoint QoS parameters.
    srx                   Display the RTPS Receiver context.
    stx                   Display the RTPS Transmitter context.
    pause                 Pause traffic.
    resume [<n>]          Resume traffic [for <n> samples].
    delay <n>             Set sleep time in ms.
    asp <d>               Assert participant.
    ase <ep>              Assert writer endpoint.
    d [<p> [<n>]]         Dump memory.
    da [<p> [<n>]]        Dump memory in ASCII.
    db [<p> [<n>]]        Dump memory in hex bytes.
    ds [<p> [<n>]]        Dump memory in hex 16-bit values.
    dl [<p> [<n>]]        Dump memory in hex 32-bit values.
    dm [<p> [<n>]]        Dump memory in mixed hex/ASCII.
    server                Start debug server.
    help                  Display general help.
    quit                  Quit main DDS program.
    exit                  Close remote connection.
```

```
>
```

Not all command characters need to be typed. Following table shows which command characters are significant and how to use them:

Table 22: Debug shell commands.

Command	Description
#	Lines starting with '#' or empty lines are seen as comments and are simply ignored.
stimer	Displays all active timers in the Technicolor DDS middleware.
sstr	Displays the string cache.
(spool | spoola)	Display memory pool usage.
scx	Display sockets that are in use.
sloc	Display the locators cache.
sdomain <d> <lf> <rf>	Display all domain data corresponding to the domain id and the local/ remote flags.
(sdisc | sdisca)	Display the local and discovered entities.
sqos	Display the QoS cache.
stopic <d> [<name>]	Display one or all Topics within the given domain.
(sendpoints | sep)	Display the Reader and Writer endpoints of all domains, local as well as discovered.
(scache | sdcache) <ep>	Display a local Reader or Writer cache.
qcache <ep> <query>	Display local Reader or Writer cache data based on the given SQL-query expression.
sproxy <ep>	Display the proxy Readers or proxy Writers corresponding with a local Writer or Reader.
seqos <ep>	Display the QoS parameters of any created entity.
srx	Display the RTPS receive message context.
stx	Display the RTPS transmitter context.
pause	Pause sending/receiving data.
resume [<n>]	Resume sending/receiving data for the given number of samples, or continously if not specified.
delay <n>	Sleep time between successive generated data samples (in milliseconds).
asp <d>	Assert manual Participant Liveliness for the specified domain.
ase <ep>	Assert Topic Liveliness for the specified endpoint.
profs	Display profiling statistics information.

Command	Description
cprof	Clear profiling statistics.
dtrace [<mode>]	Set the default tracing mode (default = no tracing) the the specified mode, or toggle the default tracing mode (none <-> all).
trace [<ep> [<mode>]]	Set the tracing mode of an endpoint (default = no tracing) to the specified mode, or toggle the tracing mode of an endpoint (only <ep> specified) or toggle the tracing mode of all endpoints (if no arguments)
cton	Start cyclic event tracing.
ctoff	Stop cyclic event tracing.
ctclr	Clear the cyclic event trace buffer.
ctmode <ctmode>	Set cyclic trace mode to cyclic (<ctmode> = 'c') or one-shot (<ctmode> = 's').
ctinfo	Display current cyclic trace settings and cyclic trace buffer filling level.
ctdump	Dump the contents of the cyclic event trace buffer.
ctsave <filename>	Save the contents of the cyclic event trace buffer in a file.
(d | da | db | ds | dl | dm) [<p> [<n>]]	Dump memory commands to dump data in ASCII, hex (8/16/32 bits) or mixed hexadecimal/ASCII formats.
server	Start a DDS Debug server for remote access.
menu	Start the program menu screen.
(help | H | ?)	Display the debug commands.
quit	Quit either the DDS Debug shell or the main program depending on the current context.
exit	Exit from a remote debug shell connection.

Resource display commands

The following DDS resources can be displayed:

- Timers. Use the `stimer` command to display the currently active timers.
- Strings. The `sstring` command can be used to display the string pool contents.
- Memory pools. The `spool` or `spoola` to display the current memory usage.
- UDP connections. The `scx` command is useful to retrieve some socket statistics.
- Locators. RTPS locators are used extensively, and the contents of the locators pool can be dumped with the `sloc` command.
- Domain entities. The domain entities can be displayed using the `sdomain`, `sdisc` and `sdisca` commands.
- QoS pools. The QoS parameters are kept in a central QoS pool, which can be displayed with the `sqos` command.

- QoS parameters. Individual QoS parameters for any DDS entity that has it can be displayed with the `seqos` command.
- Topics. Topics can be displayed in detail using the `stopic` command.
- Endpoints. The created DataReader and DataWriter endpoints can be listed with the `sendpoints` and `sep` commands.
- History cache. The DDS history cache of both DataReader and DataWriter endpoints can be displayed with the `scache` and `sdcache` commands. The actual contents of a history cache can be queried with the `qcache` command.
- RTPS Proxy contexts. RTPS Proxy Reader and Proxy Writer contexts can be shown with the `sproxy` command.
- RTPS Receiver context. The RTPS Receiver context as well as some statistics and a small error history can be dumped with the `srx` command.
- RTPS Transmitter context. Similar to `srx`, the `stx` command shows the RTPS Transmitter context.

Some example resource display commands (taken from a running shapes program, communicating with two other shapes programs) will be given, together with an explanation of what is displayed in response.

Timers

Using the `stimer` command shows the following information:

```
>stimer
DDS running for 6438.16s
# of timers started/stopped/active = 266193/0/7
# of timeouts = 106326, busy-locks = 0
Now = 694827, Timers:
 0x91919b4: time = 694835 (0.08s), user = 152639752, tcbf = 0x8049d35, name = '(null)'
 0x91924bc: time = 695323 (4.96s), user = 152642576, tcbf = 0x804a742, name = 'RED'
 0x919270c: time = 695323 (4.96s), user = 152643168, tcbf = 0x804a742, name = 'GREEN'
 0x91936dc: time = 695324 (4.97s), user = 152647216, tcbf = 0x804a742, name = 'YELLOW'
 0x918fa00: time = 695739 (9.12s), user = 152631616, tcbf = 0x80794b2, name = '(null)'
 0x919133c: time = 697741 (29.14s), user = 152638048, tcbf = 0x80680f4, name = '(null)'
 0x9192a0c: time = 697741 (29.14s), user = 152643888, tcbf = 0x80680f4, name = '(null)'
```

Strings

The `sstring` command displays the string pool:

```
>sstr

7: 'Triangle'*1
9:  04 00 00 00 52 45 44 00                              ....RED.
*1
11: 'Square'*1
13: 'SEDPbuiltinPublicationsReader'*1 'ParticipantMessageData'*1
24: 'SEDPbuiltinSubscriptionsWriter'*1
34:  07 00 00 00 59 45 4c 4c - 4f 57 00                   ....YELLOW.
*1
45: 'discoveredWriterData'*1
50: 'SEDPbuiltinPublicationsWriter'*1
52: 'ShapeType'*1
55: 'BuiltinParticipantMessageReader'*1
56: 'BuiltinParticipantMessageWriter'*1
72: 'SEDPbuiltinSubscriptionsReader'*1
87: 'SPDPdiscoveredParticipantData'*1
89: 'discoveredReaderData'*1
90: 'SPDPbuiltinParticipantReader'*1
93: 'SPDPbuiltinParticipantWriter'*1
96:  06 00 00 00 47 52 45 45 - 4e 00                      ....GREEN.
*2
```

The first number is the hash index, which is followed by the string as either ASCII characters or binary data which is displayed as mixed dump of hexadecimal numbers and ASCII characters.

As can be observed from this sample, not only ASCII strings, but binary strings as well are stored in the string pool.

The number of times a string is used in the system is denoted with the *n which follows each unique string. Here, the n parameter specifies the reference count of the string.

Pools

The `spool` command shows the following info:

```
>spool
  Name            PSize  BSize  Rsrvd    Max MPUse CPUse MXUse CXUse Alloc  NoMem
  ----            -----  -----  -----    --- ----- ----- ----- ----- -----  -----
List:
  SLIST             640     80      8      *     0     0     0     0     0      0
  SL0              3072     16    192      *    50    50     0     0     0      0
  SL1              1152     24     48      *     8     8     0     0     0      0
  SL2               384     32     12      *     0     0     0     0     0      0
  SL3               120     40      3      *     0     0     0     0     0      0
  SL4                48     48      1      *     0     0     0     0     0      0
  SL5                56     56      1      *     0     0     0     0     0      0
  SL6                64     64      1      *     0     0     0     0     0      0
  SL7                72     72      1      *     0     0     0     0     0      0
String:
  STRING            512      8     64      *    18    18     0     0     0      0
  STR_REF           592      8     74      *    18    18     0     0     0      0
Timer:
  TIMER             768     24     32      *     7     1     0     0     0      0
Buffers:
  DATA_BUF(0)      8200   8200      1      *     0     0     0     0     0      0
  DATA_BUF(1)      4104   4104      1      *     0     0     0     0     0      0
  DATA_BUF(2)      2056   2056      1      *     0     0     0     0     0      0
  DATA_BUF(3)      2064   1032      2      *     0     0     0     0     0      0
  DATA_BUF(4)      2080    520      4      *     0     0     0     0     0      0
  DATA_BUF(5)      2112    264      8      *     5     2     0     0     0      0
  DATA_BUF(6)      2176    136     16      *     0     0     0     0     0      0
  DATA_BUF(7)      2304     72     32      *     0     0     0     0     0      0
Locators:
  LOCREF            592      8     74      *    65    61     0     0     0      0
  LOCATOR          1024     32     32      *     8     8     0     0     0      0
QoS:
  QOS_REF           672      8     84      *     4     4     0     0     0      0
  QOS_DATA         1408     88     16      *     4     4     0     0     0      0
Cache:
  HIST_CACHE       2288     88     26      *    11    11     0     0     0      0
  CHANGE           1536     48     32      *     9     6     0     0     0      0
  INSTANCE         2048     64     32      *     9     8     0     0     0      0
  CCREF            1024     16     64      *    12    10     0     0     0      0
  CREF               16      8      2      *     1     1     0     0     0      0
  CWAIT             160     80      2      *     0     0     0     0     0      0
  CXFER              96     48      2      *     0     0     0     0     0      0
  XFLIST             64     32      2      *     0     0     0     0     0      0
  FILTER            128     64      2      *     0     0     0     0     0      0
  FINST              80     40      2      *     0     0     0     0     0      0
Domain:
  DOMAIN            864    864      1      1     1     1     0     0     0      0
  DPARTICIPANT     2432    304      8      *     2     2     0     0     0      0
  TYPE              240     24     10      *     5     5     0     0     0      0
  TOPIC            1280     80     16      *    10    10     0     0     0      0
  FILTER_TOPIC      128    128      1      *     0     0     0     0     0      0
  PUBLISHER         400    200      2      *     2     2     0     0     0      0
  SUBSCRIBER        384    192      2      *     2     2     0     0     0      0
  WRITER        1088    136      8      *     5     5     0     0     0      0
  READER        2688    224     12      *     6     6     0     0     0      0
  DWRITER        960     40     24      *     8     8     0     0     0      0
  DREADER       1152     48     24      *     9     9     0     0     0      0
  GUARD          160     40      4      *     0     0     0     0     0      0
DCPS:
  SAMPLE_INFO   1792     56     32      *     2     2     0     0     0      0
  WAITSET        176     88      2      *     0     0     0     0     0      0
  STATUS_COND     48     24      2      *     0     0     0     0     0      0
  READ_COND       64     32      2      *     0     0     0     0     0      0
  QUERY_COND     160     80      2      *     0     0     0     0     0      0
  GUARD_COND      48     24      2      *     0     0     0     0     0      0
  TOPIC_WAIT     128     64      2      *     0     0     0     0     0      0
RTPS:
  READER         600     40     15      *     6     6     0     0     0      0
  WRITER         528     48     11      *     5     5     0     0     0      0
  REM_READER     960     96     10      *     9     9     0     0     0      0
  REM_WRITER     880     88     10      *     8     8     0     0     0      0
  CCREF         1024     32     32      *     5     1     0     0     0      0
  MSG_BUF       4608    144     32      *     7     0     0     0     0      0
  MSG_ELEM_BUF  6656    104     64      *     7     0     0     0     0      0
UDPv4:
  UDP_CX        2640     80     33     33     5     5     0     0     0      0

Pool/max/xmax/used/xused memory = 80088/20800/0/17728/0 bytes (0 mallocs) (25%/22%)
Dynamically allocated: 2196 bytes.
malloc statistics: 38 blocks, 3672 bytes, 0 failures.
realloc statistics: 0 blocks, 0 bytes, 0 failures.
free statistics: 16 blocks, 1476 bytes.
Dynamic pool block stats: <=64K - max/used/msize/size: 0/0/0/0
                           >64K - max/used           : 0/0
Total heap memory: 80088 bytes.
```

The display first shows some columns, followed by summary information.

The first column shows either the origin (ex: RTPS:) or the name of the memory pool:

Table 23: Types of memory blocks

Name	Description
SLIST	Skiplist list.
SL0..7	Skiplist node containing i*2 forward pointers. Ex.: SL3 contains 6 forward pointers.
STRING	String descriptor. Actual string data is stored in a separate string heap.
STR_REF	Reference to a string descriptor.
TIMER	Timer node.
DATA_BUF(0..7	Data buffer pools having room for 64 up to 8K bytes as subsequent powers of 2, e.g. 64, 128, 256, etc. Buffers have a reference count and can be referenced from multiple readers and writers.
LOCREF	Reference to a locator node.
LOCATOR	A locator node contains locator storage and a reference count.
QOS_REF	Reference to a QoS node.
QOS_DATA	A QoS node contains Quality of Service parameters in a unified format, which is suitable for DataWriters, DataReaders and Topics either locally or remote.

Name	Description
HIST_CACHE	History cache descriptor as used in both DataWriters and DataReaders. It describes all data-related handling, and contains a list of samples, and possibly a list of instances.
CHANGE	A Change descriptor typically contains a reference to actual change data (handle, data stored in a DATA_BUF or allocated directly).
INSTANCE	Each instance, e.g. key-dependent data is a descriptor that contains a list of samples as well as various state information.
CCREF	The Change reference descriptor is used while queueing sample data. Keyed samples typically have two change references. Non-keyed is typically a single change reference.
CREF	The Reader Cache reference is used in order to chain matched local DataReaders for a local DataWriter.
CWAIT	The Cache Wait context is used when an application needs to wait until some condition needs to be fulfilled, such as room for writing a new sample, wait for acknowledgements, etc.
CXFER	Samples that need to be sent to local DataReader caches from a local DataWriter cache, but which can not be transferred yet, due to the destination cache having no room, are queued using Cache Transfer descriptors.
XFLIST	A list of Cache Transfer descriptors to be received when the local DataReader cache accepts new samples.
FILTER	The Time-Based filter descriptor is used for a local Reader cache to limit the receive rate from a specific Writer.
FINST	This a per-instance descriptor that contains a delayed sample for reception.
DOMAIN	A complete DomainParticipant descriptor.
DPARTICIPANT	A Discovered, i.e. remote DomainParticipant.
TYPE	Type descriptor.
TOPIC	Topic descriptor.
FILTER_TOPIC	A Filtered topic is derived from a normal topic.
PUBLISHER	Publisher descriptor.
SUBSCRIBER	Subscriber descriptor.
WRITER	A DDS DataWriter, suitable for local and remote use, referring to a history cache, optionally extended with an RTPS Writer context.
READER	A DDS DataReader, suitable for local and remote use, referring to a history cache, optionally extended with an RTPS Reader context.
DWRITER	A Discovered DataWriter.
DREADER	A Discovered DataReader.
GUARD	Guard contexts are used to implement various QoS parameters, such as Liveliness, Deadline, Lifespan and DataReader Lifetime QoS.

Name	Description
SAMPLE_INFO	The SampleInfo structure is allocated when using any of the various DDS_DataReader_read/take() functions, until a DDS_DataReader_return_loan() is called.
WAITSET	A WaitSet context.
STATUS_COND	A StatusCondition descriptor.
READ_COND	A ReadCondition descriptor.
QUERY_COND	A QueryCondition descriptor.
GUARD_CONI	A GuardCondition descriptor.
TOPIC_WAIT	Whenever the application needs to wait on a remotely defined topic via DDS_DomainParticipant_find_topic(), this context is used.
RTPS::READER	When a local DataReader is matched for the first time with a remote DataWriter, the RTPS Reader context is added to the DataReader in order to manage the RTPS Proxy Writers.
RTPS::WRITER	When a local DataWriter is matched for the first time with a remote DataReader, the RTPS Reader context is added to the DataWriter in order to manage the RTPS Proxy Readers.
REM_READER	An RTPS Proxy Reader descriptor is used for each matched remote DataWriter.
REM_WRITER	An RTPS Proxy Writer descriptor is used for each matched remote DataReader.
RTPS::CCREF	When data samples are queued in RTPS, whether for a Proxy Reader or for a Proxy Writer, they are queued using Cache Change Reference descriptors, that point to cloned Change descriptors.
MSG_BUF	An RTPS Message descriptor is used, both when RTPS messages are received, and when they are under construction, i.e. when RTPS Submessages are added to them.
MSG_ELEM_BL	An RTPS Submessage descriptor is used both for Submessage headers, as well as for extra data that needs to be concatenated to a submessage.
UDP_CX	A UDP Socket descriptor. One of these is used for sending, for each remote peer locator a receiving socket is used.

This is then followed by a number of columns specifying either a number of bytes or a counter:

Table 24: `spool` and `spoola` counters

Column	Description
Addr	The memory address of the preallocated pool.
PSize	The size of the preallocated pool.
BSize	Block size of each element in the pool.
Rsrvd	Number of reserved elements.

Column	Description
Max	Maximum number of blocks that may be allocated of the pool (or * if unlimited).
MPUse	Maximum number of blocks ever in use during the program lifetime.
CPUse	Current number of blocks in use by the program.
MXUse	Maximum number of blocks ever dynamically allocated.
CXUse	Current number of blocks that are dynamically allocated.
Alloc	Total number of dynamic allocations.
Nomem	Total number of unsuccessful allocations (i.e. out-of-memory conditions).
Block	First available pool block.

If the DDS library was compiled with the FORCE_MALLOC define, which disables preallocated pools, then a number of fields will not be significant, such as Addr, PSize, Rsrvd, MPUse, CPUse and Block and these will then be set to 0 (or NULL).

As can be seen from this example, only approximately 20 KBytes of memory was ever used, but the preallocated pools consume more than 80 KBytes.

Optimizing memory usage can be done very easily by looking at both the MPUse and MXUse fields, which, when added together, indicate the total number of blocks ever used in the program. Setting the preallocated pool requirements then allows to allocate everything from that pool and will no longer use malloc() for those pool elements.

Preallocated pools have a number of advantages compared to pure malloc() and free() functions:

1. Allocation and release functions are at least twice as fast compared to dynamic memory allocation functions.
2. There is no memory fragmentation overhead (saves ~20%).
3. Pool usage is deterministic and never leads to unexpected out-of-memory conditions while using less elements than the pool sizes permit, since all memory blocks will be allocated at startup.
4. Pool usage can be restricted in order to limit the number of memory resources that a DDS program is allowed to use.

Using the FORCE_MALLOC define is alternative way of handling memory, which results in ALL memory being allocated via malloc(), effectively bypassing the preallocated pool mechanism.

Sockets

The `scx` command shows the UDP resources:

```
>scx
Sending socket:
  0.0.0.0:0            fd: 3, id: 0, errors: 0, empty: 0, too_short: 0
  no buffers: 0, octets Tx/Rx: 4839384/0, packets Tx/Rx: 47603/0
Locators:
  192.168.64.128:7425  fd: 6, id: 1, errors: 0, empty: 0, too_short: 0
  no buffers: 0, octets Tx/Rx: 0/8398788, packets Tx/Rx: 0/70777
  239.255.0.1:7401     fd: 7, id: 1, errors: 0, empty: 0, too_short: 0
  no buffers: 0, octets Tx/Rx: 0/3230768, packets Tx/Rx: 0/23364
  192.168.64.128:7424  fd: 8, id: 1, errors: 0, empty: 0, too_short: 0
  no buffers: 0, octets Tx/Rx: 0/3748, packets Tx/Rx: 0/35
  239.255.0.1:7400     fd: 9, id: 1, errors: 0, empty: 0, too_short: 0
  no buffers: 0, octets Tx/Rx: 0/118948, packets Tx/Rx: 0/439
```

Where errors, empty, too_short and no_buffers indicate respectively the number of errors that occurred, the number of unexpted empty reads, too short messages and times there was not enough memory to handle reception properly.

The octets and packets statistics can be useful to determine how much traffic was present for individual peers.

Locators

The `sloc` command gives the following output:

```
>sloc
UDP:239.255.0.1:7400*24
UDP:239.255.0.1:7401*7
UDP:192.168.64.128:7414*7
UDP:192.168.64.128:7415*3
UDP:192.168.64.128:7422*7
UDP:192.168.64.128:7423*3
UDP:192.168.64.128:7424*9
UDP:192.168.64.128:7425*1
```

Each locator as displayed as:

- Protocol type (only UDP is used here).
- The protocol specific address (unicast or multicast IP address).
- A protocol specific port number (:port).
- How many times the locator is referenced (the *n).

As can be seen from the display, locators are reused a lot. Some of them, like the multicast data locator even 24 times.

Quality of Service

Concept definition.

The `sqos` command shows the following output:

```
>sqos
5: 0x9b865b0*2
7: 0x9b86660*5
23: 0x9b86608*22
33: 0x9b866b8*2
```

The QoS display is somewhat simplistic because it simply displays the address of the in-memory unified QoS data container, following the hash key and followed by the reference counter.

It does however show that reusing QoS parameters is very advantageous in order to reduce the overall memory footprint.

Displaying specific entity QoS parameters is much more appropriate, as can be done with the `seqos` command.

In order to do this, the user needs to specify the handle of the entity that needs to be displayed:

```
>seq 42
Reader QoS:
    USER_DATA: <empty>
    DURABILITY: TRANSIENT_LOCAL
    DEADLINE: <inf>
    LATENCY_BUDGET: 0s
    OWNERSHIP: SHARED
    LIVELINESS: AUTOMATIC, lease_duration=<inf>
    RELIABILITY: RELIABLE, max_blocking_time=0.100000000s
    DESTINATION_ORDER: BY_RECEPTION_TIMESTAMP
    HISTORY: KEEP_LAST, depth=1
    RESOURCE_LIMITS: max_samples=<inf>, max_inst=<inf>, max_samples_per_inst=<inf>
    TIME_BASED_FILTER: minimum_separation=0s
    READER_DATA_LIFECYCLE: autopurge_nowriter=<inf>, autopurge_disposed=<inf>
G:  PARTITION: <none>
    GROUP_DATA: <empty>
T:  TOPIC_DATA: <empty>
>seq 45
Writer QoS:
    USER_DATA: <empty>
    DURABILITY: TRANSIENT_LOCAL
    DURABILITY_SERVICE:
    DEADLINE: <inf>
    LATENCY_BUDGET: 0s
    OWNERSHIP: SHARED
    OWNERSHIP_STRENGTH: 0
    LIVELINESS: AUTOMATIC, lease_duration=<inf>
    RELIABILITY: RELIABLE, max_blocking_time=0.100000000s
    TRANSPORT_PRIORITY: 0
    LIFESPAN: <inf>
    DESTINATION_ORDER: BY_RECEPTION_TIMESTAMP
    HISTORY: KEEP_LAST, depth=1
    RESOURCE_LIMITS: max_samples=<inf>, max_inst=<inf>, max_samples_per_inst=<inf>
    WRITER_DATA_LIFECYCLE: autodispose_unregistered_instances=1
G:  PARTITION: <none>
    GROUP_DATA: <empty>
T:  TOPIC_DATA: <empty>
```

The G: and T: references in the display refer to data that is effectively present in the Group (Partition QoS and Group Data) and Topic (Topic Data) respectively.

This command is not restricted to DataReader and DataWriter QoS. It can be used to display the QoS parameters of any entity in the DDS domains that has QoS parameters. It is just a matter of giving the correct instance handle.

Domain data

There are three commands for displaying domain data:

- The `sdomain` command is the most versatile, but requires some arguments.
- The `sdisc` and `sdisca` commands don't expect arguments. The `sdisc` command shows minimal info, the `sdisca` command shows most available info.

The sdomain command accepts 3 arguments:

- The domain identifier.
- The local information to display as a hexadecimal bitmap.
- The discovered information to display as a hexadecimal bitmap.

The bitmap has following values:

Table 25: Domain Information bitmap flags

Flag	Description
1	Include Locators
2	Include Builtin endpoints
4	Include Types.
8	Include Topics.
16	Include Guard records.
32	Include Peer participants.

Some examples:

```
>sdisc
Domain 0 (pid=7): {1}
    GUID prefix: c0a84080:00072816:03e80000
    RTPS Protocol version: v2.1
    Vendor Id: 1.9 - Technicolor
    Expects Inline QoS: no
    Enabled: 1
    Meta Unicast:
        UDP:192.168.64.128:7424
    Meta Multicast:
        UDP:239.255.0.1:7400
    Default Unicast:
        UDP:192.168.64.128:7425
    Default Multicast:
        UDP:239.255.0.1:7401
    Manual Liveliness: 0
    Lease duration: 30.000000000s
    Endpoints: 11 entries (6 readers, 5 writers).
    Resend period: 10.000000000s
    Destination Locators:
        UDP:239.255.0.1:7400
    Discovered participants:
      Peer #0: {27}
        GUID prefix: c0a84080:00021ed3:03e80000
        RTPS Protocol version: v2.1
        Vendor Id: 1.9 - Technicolor
        Expects Inline QoS: no
        Ignored: 0
        Meta Unicast:
            UDP:192.168.64.128:7414
        Meta Multicast:
            UDP:239.255.0.1:7400
        Default Unicast:
            UDP:192.168.64.128:7415
        Default Multicast:
            UDP:239.255.0.1:7401
        Manual Liveliness: 0
        Lease duration: 30.000000000s
        Endpoints: 8 entries (4 readers, 4 writers).
        Timer = 29.47s
      Peer #1: {20}
        GUID prefix: c0a84080:00062810:03e80000
        RTPS Protocol version: v2.1
        Vendor Id: 1.9 - Technicolor
        Expects Inline QoS: no
        Ignored: 0
        Meta Unicast:
            UDP:192.168.64.128:7422
        Meta Multicast:
            UDP:239.255.0.1:7400
        Default Unicast:
            UDP:192.168.64.128:7423
        Default Multicast:
            UDP:239.255.0.1:7401
        Manual Liveliness: 0
        Lease duration: 30.000000000s
        Endpoints: 9 entries (5 readers, 4 writers).
        Timer = 29.38s

>sdisca
Domain 0 (pid=7): {1}
    GUID prefix: c0a84080:00072816:03e80000
    RTPS Protocol version: v2.1
    Vendor Id: 1.9 - Technicolor
    Expects Inline QoS: no
    Enabled: 1
    Meta Unicast:
        UDP:192.168.64.128:7424
    Meta Multicast:
        UDP:239.255.0.1:7400
    Default Unicast:
        UDP:192.168.64.128:7425
    Default Multicast:
        UDP:239.255.0.1:7401
    Manual Liveliness: 0
    Lease duration: 30.000000000s
    Endpoints: 11 entries (6 readers, 5 writers).
        000001-7, {42}, InlineQoS: No, Reader, Square/ShapeType
        000002-7, {43}, InlineQoS: No, Reader, Triangle/ShapeType
        000003-2, {45}, InlineQoS: No, Writer, Square/ShapeType
    Topics:
        BuiltinParticipantMessageReader/ParticipantMessageData
        BuiltinParticipantMessageWriter/ParticipantMessageData
        SEDPbuiltinPublicationsReader/discoveredWriterData
        SEDPbuiltinPublicationsWriter/discoveredWriterData
        SEDPbuiltinSubscriptionsReader/discoveredReaderData
        SEDPbuiltinSubscriptionsWriter/discoveredReaderData
        SPDPbuiltinParticipantReader/SPDPdiscoveredParticipantData
        SPDPbuiltinParticipantWriter/SPDPdiscoveredParticipantData
        Square/ShapeType Triangle/ShapeType
    Resend period: 10.000000000s
    Destination Locators:
        UDP:239.255.0.1:7400
    Discovered participants:
      Peer #0: {27}
        GUID prefix: c0a84080:00021ed3:03e80000
        RTPS Protocol version: v2.1
        Vendor Id: 1.9 - Technicolor
        Expects Inline QoS: no
        Ignored: 0
        Meta Unicast:
            UDP:192.168.64.128:7414
        Meta Multicast:
            UDP:239.255.0.1:7400
        Default Unicast:
            UDP:192.168.64.128:7415
        Default Multicast:
            UDP:239.255.0.1:7401
        Manual Liveliness: 0
        Lease duration: 30.000000000s
        Endpoints: 8 entries (4 readers, 4 writers).
            000001-2, {40}, InlineQoS: No, Writer, Square/ShapeType
            000002-7, {39}, InlineQoS: No, Reader, Square/ShapeType
        Topics:
            Square/ShapeType
        Timer = 26.04s
      Peer #1: {20}
        GUID prefix: c0a84080:00062810:03e80000
        RTPS Protocol version: v2.1
        Vendor Id: 1.9 - Technicolor
        Expects Inline QoS: no
        Ignored: 0
        Meta Unicast:
            UDP:192.168.64.128:7422
        Meta Multicast:
            UDP:239.255.0.1:7400
        Default Unicast:
            UDP:192.168.64.128:7423
        Default Multicast:
            UDP:239.255.0.1:7401
        Manual Liveliness: 0
        Lease duration: 30.000000000s
        Endpoints: 9 entries (5 readers, 4 writers).
            000001-7, {34}, InlineQoS: No, Reader, Square/ShapeType
            000002-7, {36}, InlineQoS: No, Reader, Triangle/ShapeType
            000003-2, {38}, InlineQoS: No, Writer, Triangle/ShapeType
        Topics:
            Square/ShapeType Triangle/ShapeType
        Timer = 25.95s
```

Instance handles are displayed as {n}, where n is a decimal number. This number can be referenced when using other commands such as `seqos` and `sdcache` (see further).

As can be seen in this example, the shapes program has discovered two remote participants, with the first having a square writer and a square reader, and the other having both square and triangle readers and a triangle writer.

Topics

Topics can either be displayed as the list of Topics in the domain, or more specific Topic information can be shown.

The `stopic` command expects a Domain Identifier, and then displays all Topics in the Domain as follows:

```
>stopic
?number expected!
BuiltinParticipantMessageReader/ParticipantMessageData:
    # of local create/find_topic() calls = 1, # of discoveries = 0
    Writers:
        DW: {29}    c0a84080:000208e8:03e80000-000200-c2
        DW: {22}    c0a84080:000108e6:03e80000-000200-c2
    Readers:
        R:  {17}    c0a84080:000008e2:03e80000-000200-c7
BuiltinParticipantMessageWriter/ParticipantMessageData:
    # of local create/find_topic() calls = 1, # of discoveries = 0
    Writers:
        W:  {19}    c0a84080:000008e2:03e80000-000200-c2
    Readers:
        DR: {28}    c0a84080:000208e8:03e80000-000200-c7
        DR: {21}    c0a84080:000108e6:03e80000-000200-c7
SEDPbuiltinPublicationsReader/discoveredWriterData:
    # of local create/find_topic() calls = 1, # of discoveries = 0
    Writers:
        DW: {31}    c0a84080:000208e8:03e80000-000003-c2
        DW: {24}    c0a84080:000108e6:03e80000-000003-c2
    Readers:
        R:  {9}     c0a84080:000008e2:03e80000-000003-c7
SEDPbuiltinPublicationsWriter/discoveredWriterData:
    # of local create/find_topic() calls = 1, # of discoveries = 0
    Writers:
        W:  {13}    c0a84080:000008e2:03e80000-000003-c2
    Readers:
        DR: {30}    c0a84080:000208e8:03e80000-000003-c7
        DR: {23}    c0a84080:000108e6:03e80000-000003-c7
SEDPbuiltinSubscriptionsReader/discoveredReaderData:
    # of local create/find_topic() calls = 1, # of discoveries = 0
    Writers:
        DW: {33}    c0a84080:000208e8:03e80000-000004-c2
        DW: {26}    c0a84080:000108e6:03e80000-000004-c2
    Readers:
        R:  {11}    c0a84080:000008e2:03e80000-000004-c7
SEDPbuiltinSubscriptionsWriter/discoveredReaderData:
    # of local create/find_topic() calls = 1, # of discoveries = 0
    Writers:
        W:  {15}    c0a84080:000008e2:03e80000-000004-c2
    Readers:
        DR: {32}    c0a84080:000208e8:03e80000-000004-c7
        DR: {25}    c0a84080:000108e6:03e80000-000004-c7
SPDPbuiltinParticipantReader/SPDPdiscoveredParticipantData:
    # of local create/find_topic() calls = 1, # of discoveries = 0
    Writers: <none>
    Readers:
        R:  {7}     c0a84080:000008e2:03e80000-000100-c7
SPDPbuiltinParticipantWriter/SPDPdiscoveredParticipantData:
    # of local create/find_topic() calls = 1, # of discoveries = 0
    Writers:
        W:  {5}     c0a84080:000008e2:03e80000-000100-c2
    Readers: <none>
Square/ShapeType:
    # of local create/find_topic() calls = 1, # of discoveries = 3
    Writers:
        W:  {46}    c0a84080:000008e2:03e80000-000003-2
        DW: {34}    c0a84080:000108e6:03e80000-000001-2
    Readers:
        R:  {43}    c0a84080:000008e2:03e80000-000001-7
        DR: {37}    c0a84080:000208e8:03e80000-000001-7
        DR: {36}    c0a84080:000108e6:03e80000-000002-7
Triangle/ShapeType:
    # of local create/find_topic() calls = 1, # of discoveries = 3
    Writers:
        DW: {40}    c0a84080:000208e8:03e80000-000003-2
    Readers:
        R:  {44}    c0a84080:000008e2:03e80000-000002-7
        DR: {41}    c0a84080:000108e6:03e80000-000003-7
        DR: {38}    c0a84080:000208e8:03e80000-000002-7
```

If no Domain Identifier is specified, the debugger gives a warning, but still continues listing the Topics in the first Domain.

The following information is displayed for each Topic:

- Topic name / type.
- Some Topic parameters needed to do correct Topic accounting.
- The list of Writers (local and discovered) associated with the Topic.
- The list of Readers (local and discovered) associated with the Topic.

The following information is displayed for each Writer or Reader:

- The type of endpoint (W = local Writer, DW = discovered Writer, R = local Reader and DR = discovered Reader).
- The handle of the endpoint (between '{' and '}').
- The unique GUID.

When the Topic name is specified, a lot more information is given, as shown below:

```
>stopic 0 Square
Square/ShapeType:
    Preferred: CDR, dynamic: 0, length: 140, keys: 1, mkeysize: 133
    struct ShapeType {
        string<128> color;  //@key
        long x;
        long y;
        long shapesize;
    };
    # of local create/find_topic() calls = 1, # of discoveries = 3
    Writers:
        W:  {45}    c0a84080:000008cc:03e80000-000003-2
        DW: {36}    c0a84080:0001091e:03e80000-000002-2
    Readers:
        R:  {42}    c0a84080:000008cc:03e80000-000001-7
        DR: {37}    c0a84080:00020973:03e80000-000001-7
        DR: {34}    c0a84080:0001091e:03e80000-000001-7
```

As can be seen from this example, for individual topics, extra Topic type information is given:

- The on-the-wire encoding/decoding type (CDR in this case).
- Whether the type is dynamic or fixed-size.
- The size of a sample in bytes.
- Whether or not key information is present, and the total size of the key fields.
- The type itself as originally registered as an IDL type definition.

For filtered topics, if displayed with this command, even more data is displayed, e.g. the expression string, expression parameters and the generated bytecode program that is used for filtering data.

In the following example, a filtered topic is shown, as part of a topics list:

```
Triangle/ShapeType:
    Filter name: Filter_2
    Class name: DDSSQL
    Expression: x > %0 and x < %1 and y > %2 and y < %3
    Parameters:
        %0 = "50"
        %1 = "150"
        %2 = "30"
        %3 = "90"
    Bytecode program size: 38 bytes.
    # of local create/find_topic() calls = 1, # of discoveries = 0
    Writers: <none>
    Readers:
        R:  {56} 0a000044:000c38a9:03e80000-000002-7
```

To show the filter topic in detail, the filter name needs to be specified as the topic name:

```
>stopic 0 Filter_2
Triangle/ShapeType:
    Filter name: Filter_2
    Class name: DDSSQL
    Expression: x > %0 and x < %1 and y > %2 and y < %3
    Parameters:
        %0 = "50"
        %1 = "150"
        %2 = "30"
        %3 = "90"
    Bytecode program:
  0000        LDWS 1
  0003        LPWS 0
  0005        CMPWS
  0006        BLE 37
  0009        LDWS 1
  0012        LPWS 1
  0014        CMPWS
  0015        BGE 37
  0018        LDWS 2
  0021        LPWS 2
  0023        CMPWS
  0024        BLE 37
  0027        LDWS 2
  0030        LPWS 3
  0032        CMPWS
  0033        BGE 37
  0036        RETT
  0037        RETF
    Preferred: CDR, dynamic: 0, length: 140, keys: 1, mkeysize: 133
    struct ShapeType {
        string<128> color;  //@key
        long x;
        long y;
        long shapesize;
    };
    # of local create/find_topic() calls = 1, # of discoveries = 0
    Writers: <none>
    Readers:
        R:  {56} 0a000044:000c38a9:03e80000-000002-7
```

For discovered filtered topics, where the filter is specified by a remote Reader, the same information will also be present locally, and can also be displayed in this manner.

Endpoints

A list of endpoints and some condensed endpoint information can be displayed with the sendpoints and sep commands as shown here:

```
>sep
DCPS:
    0/000001-07 R{42}   0/2    Square/ShapeType
    0/000002-07 R{43}   0/1    Triangle/ShapeType
    0/000003-02 W{45}   1/1    Square/ShapeType
    0/000003-c2 W{13}   1/1    SEDPbuiltinPublicationsWriter/discoveredWriterData
    0/000003-c7 R{9}    0/0    SEDPbuiltinPublicationsReader/discoveredWriterData
    0/000004-c2 W{15}   2/2    SEDPbuiltinSubscriptionsWriter/discoveredReaderData
    0/000004-c7 R{11}   0/0    SEDPbuiltinSubscriptionsReader/discoveredReaderData
    0/000100-c2 W{5}    1/1    SPDPbuiltinParticipantWriter/SPDPdiscoveredParticipantData
    0/000100-c7 R{7}    0/0    SPDPbuiltinParticipantReader/SPDPdiscoveredParticipantData
    0/000200-c2 W{19}   0/0    BuiltinParticipantMessageWriter/ParticipantMessageData
    0/000200-c7 R{17}   0/0    BuiltinParticipantMessageReader/ParticipantMessageData
RTPS:
    0/000001-07 R{42}   0/2    Square/ShapeType
                ->    GUID: c0a84080:00021ed3:03e80000-000001-2
    0/000002-07 R{43}   0/1    Triangle/ShapeType
                ->    GUID: c0a84080:00062810:03e80000-000003-2
    0/000003-02 W{45}   0/1    Square/ShapeType
                ->    GUID: c0a84080:00021ed3:03e80000-000002-7
                ->    GUID: c0a84080:00062810:03e80000-000001-7
    0/000003-c2 W{13}   0/1    SEDPbuiltinPublicationsWriter/discoveredWriterData
                ->    GUID: c0a84080:00062810:03e80000-000003-c7
                ->    GUID: c0a84080:00021ed3:03e80000-000003-c7
    0/000003-c7 R{9}    0/0    SEDPbuiltinPublicationsReader/discoveredWriterData
                ->    GUID: c0a84080:00062810:03e80000-000003-c2
                ->    GUID: c0a84080:00021ed3:03e80000-000003-c2
    0/000004-c2 W{15}   0/2    SEDPbuiltinSubscriptionsWriter/discoveredReaderData
                ->    GUID: c0a84080:00062810:03e80000-000004-c7
                ->    GUID: c0a84080:00021ed3:03e80000-000004-c7
    0/000004-c7 R{11}   0/0    SEDPbuiltinSubscriptionsReader/discoveredReaderData
                ->    GUID: c0a84080:00062810:03e80000-000004-c2
                ->    GUID: c0a84080:00021ed3:03e80000-000004-c2
    0/000100-c2 W{5}    1/1    SPDPbuiltinParticipantWriter/SPDPdiscoveredParticipantData
                ->    Locator: UDP:239.255.0.1:7400
    0/000100-c7 R{7}    0/0    SPDPbuiltinParticipantReader/SPDPdiscoveredParticipantData
    0/000200-c2 W{19}   0/0    BuiltinParticipantMessageWriter/ParticipantMessageData
                ->    GUID: c0a84080:00062810:03e80000-000200-c7
                ->    GUID: c0a84080:00021ed3:03e80000-000200-c7
    0/000200-c7 R{17}   0/0    BuiltinParticipantMessageReader/ParticipantMessageData
                ->    GUID: c0a84080:00062810:03e80000-000200-c2
                ->    GUID: c0a84080:00021ed3:03e80000-000200-c2
```

Endpoints are displayed as two groups of endpoints.

The first group lists the endpoints as known on DCPS level. This typically includes all endpoints that are defined in the domain without further information.

The second group lists the endpoints that have RTPS connectivity with discovered endpoints. For each of those, either the ReaderLocator instance (SPDPbuiltinParticipantWriter) or the Proxy Reader or Writer contexts are shown as GUIDs.

Endpoints are displayed as:

- A a Domain Id/Entity Id pair, followed by
- the endpoint type (R or W) and handle,
- summary history cache information as <*number_of_samples*> / <*number_of_instances*>, followed by
- Topic name / Type name strings.

In the RTPS group, each matched endpoint is listed explicitly on following lines.

History Cache

The history cache contents can be shown with the `scache` or `sdcache` commands as shown below:

```
>sdc 42
Topic/Type: Square/ShapeType
GUID: c0a84080:000a38b9:03e80000-000001-7
Max. depth = 1, Flags = Transient-Local/Auto-dispose/Inst-ordered/Keep-last
Limits: * samples, * instances, 1 samples/instance
Hashed: 0, monitor: 0, inform: 1, unacked: 0, ownership: shared
Changes:
        42.998126224s - [0x80ec7c8*2:ALIVE:0:h=2,0.1;R,NN]->0x80e9ed8*2,144 bytes
        43.119194585s - [0x80ec888*2:ALIVE:0:h=1,0.1;NR,N]->0x80ea114*1,144 bytes
Last handle = 2, Maximum key size = 133
   1: 30727bee:ff213066:4d6cfc73:0a6c72c8, ALIVE, NOT_NEW, D/NW=0/1, I, , W: 40
      Key:
        04 00 00 00 52 45 44 00                            ....RED.
        -: (1){ 43.119194585s - [0x80ec888*2:ALIVE:0:h=1,0.1]->0x80ea114*1,144 bytes
}
   2: 62dc371a:f8c3030a:82f1e817:d01e2c19, ALIVE, NOT_NEW, D/NW=0/1, , W: 45
      Key:
        07 00 00 00 59 45 4c 4c - 4f 57 00                 ....YELLOW.
        -: (1){ 42.998126224s - [0x80ec7c8*2:ALIVE:0:h=2,0.1]->0x80e9ed8*2,144 bytes
}
```

In this particular example, where we display the contents of the Square DataReader, samples remain in the history cache, since we always read() from the cache while processing the data instead of doing a take().

The following information is effectively displayed:

- The Topic/Type names.
- The GUID of the DataReader.
- The maximum depth for *KEEP_LAST* caches or the *max_samples_per_instance* limit for *KEEP_ALL* caches.
- Cache flags that are derived from QoS parameters and which control cache behaviour.

- The resource limits for this cache.
- Whether the cache is hashed, actively being monitored.
- The number of as yet unacknowledged samples.
- The type of ownership.
- The global list of changes in the cache, sorted in arrival order (2 samples in this cache).
- If the cache has multiple instances, i.e. is keyed, the last handle and the key size are displayed, as well as the instance list.
- Each active instance is listed as:

1. The instance handle.
2. The hash value of the key as an MD5 checksum if the key is larger than 16 bytes, or the key itself.
3. The last instance and view states.
4. The number of disposed/no_writer state changes.
5. The handle of the writers actively writing data to this cache.
6. The key value if the key size is larger than 16 bytes.
7. The arrival time of the last sample.
8. The list of samples queued in that instance.

The instance list for a multi-instance cache is either a linear list, if small enough, or a dual skiplist, where the first skiplist has as key the keyhash value, and the second skiplist has as key the instance handle.

Multi-instance caches are automatically converted from one format to the other, when items are added or removed from.

For each sample, the following information is displayed:

- The timestamp for when the sample was generated, relative since the time the program was started, followed by " - ["
- The address of the Change descriptor and the number of references to it.
- If the change is ALIVE or DISPOSED or UNREGISTERED or ZOMBIE (both disposed and unregistered).
- The number of outstanding acknowledgements.
- The writer handle.
- The sequence number as a *high.low* tuple.
- If the sample was due to a directed write, the list of destinations (after a '=>' symbol).
- If a reader cache (in this case), the sample state (NR=Not Read, R=Read) and the view state (NN=Not New, N=New), followed by ']'
- If the sample contains data (ALIVE sample), this is indicated with a "-><addr>*<nrefs>,<len> bytes" string where:

1. <addr> is the memory address of the sample data.
2. <nrefs> is the number of references to this sample data.
3. <len> is the total length of the sample data in bytes.

The cache contents can be queried with the qcache command, as follows:

```
>qcache 42 x < 150 and y > 10
1: {color="GREEN", x=104, y=83, shapesize=30}
```

Proxy Endpoints

The `sproxy` command can be used to display RTPS proxy endpoint contexts:

```
>sproxy 45
GUID: c0a84080:00021ed3:03e80000-000002-7 - InlineQoS=0, Reliable=1, Active=1
    States (Control/Tx/Ack): READY/IDLE/WAITING
    Changes: : (0)
    LastAck=2682
    Unicast reply locator:
        UDP:192.168.64.128:7415
GUID: c0a84080:00062810:03e80000-000001-7 - InlineQoS=0, Reliable=1, Active=1
    States (Control/Tx/Ack): READY/IDLE/WAITING
    Changes: : (0)
    LastAck=354304
    Unicast reply locator:
        UDP:192.168.64.128:7423
```

As shown above, for the local DataWriter with handle 45, there are two proxy Reader contexts.

For each proxy context, the following information is displayed:

- The GUID
- The type of endpoint, e.g. reliable or best-effort.
- The proxy state machine states (see RTPS specification for details).
- The queued changes.
- The last acknowledgement number.
- The determined locator to use for replies.

RTPS Send/Receive contexts

The RTPS Send and Receive contexts can be displayed with the `stx` and `srx` commands respectively.

Following examples should demonstrate this:

```
>stx
# of locator transmits: 30761
# of missing locator events: 0
# of out-of-memory conditions: 0

>srx
Protocol version: {2, 1}
Src. Vendor Id: {0x1, 0x9}
Src. GUID Prefix: c0a84080:00062810:03e80000
Dst. GUID Prefix: c0a84080:000d3c04:03e80000
Domain: 0x9644b00
# of invalid submessages: 0
# of too short submessages: 0
# of invalid QoS errors: 0
# of out-of-memory errors: 0
# of unknown destination errors: 15108
# of invalid marshalling data: 0
Last error: Unknown destination
Last MEP:
      00 00 00 00 d8 0e 64 09 - d0 0e 64 09 bc 00 04 00   ......d...d.....
      15 07 bc 00 00 00 02 07 - 00 00 03 02 00 00 00 00   ................
      9e 1d 00 00 00 00 00 00 - 9e 1d 00 00 6e 1e 00 00   ............n...
      42 b9 f5 a5 a1 18 6f 2c - 5c de 78 d5 01 00 00 00   B.....o,\.x.....
      00 00 00 00 e8 10 64 09 - 00 00 00 00 90 00 0a 00   ......d.........
      00 00 00 00 00 00 00 00 - c8 37 64 09 00 00 00 00   .........7d.....
      00 00 00 00                                         ....
Last domain: 0
Last msg:
      00 00 10 00 00 00 00 00 - 00 00 03 02 00 00 00 00   ................
      9e 1d 00 00                                         ....
```

Memory dumps

A number of commands are available for dumping memory regions in a variety of formats.

The following commands can be used for this:

- `d` or `dm` display memory in mixed hexadecimal and ASCII characters.
- `db` displays memory as a number of bytes in rows of 16 bytes.
- `ds` displays memory as a number of short words (16-bits) in rows of 8 words.
- `dl` displays memory as a number of long words (32-bits) in rows of 4 long words.

If no arguments are given with a command, the dump continues from the memory address that follows the previous dump command, and dumps 64 bytes, e.g. 4 rows by default.

Make sure that the first dump command that is given specifies a valid address, since the default memory location is set to 0 if no dump command was specified yet!

If an address is specified, dumping starts from that location.

If a number follows the address, it indicates the length (in bytes) of the memory region that needs to be dumped, otherwise the default 64-byte range is dumped.

Be very careful with these commands. If an inappropriate address is specified, segfaults might occur!!

Control commands

The DDS Debug Shell provides a number of commands that can be used to control application programs directly:

- The `pause`, `resume`, `delay`, `menu` and `quit` commands can be used, for example, to control the `dcps` test program and the `shapes` demo program. This is described in the chapters on these programs. Other application programs can register themselves in the Shell to enable some of these commands selectively (see dds/dds_debug.h).
- The `server` command can be used to start the DDS Debug server in order to provide remote access to the shell via the `telnet` protocol.
- Remotely connected telnet sessions should be closed via the `exit` command.
- The `ase` and `asp` commands can be used to assert Topic and Participant Liveliness, but should only be used in the context of the `dcps` test program.

Tracing and Profiling commands

Various classes of tracing and profiling are supported in Technicolor DDS:

- RTPS Messages, Finite State Machine state changes, Control and Timer tracing can be used while debugging (DDS Debugging must be enabled for this) in order to verify correct operation of selective endpoints. The `dtrace` and `trace` commands are provided for this.
- Cyclic tracing can optionally be enabled via an additional compilation option (-DCTRACE_USED) to provide for DDS-wide event tracing via the `cton`, `ctoff`, `ctclr`, `ctmode`, `ctinfo`, `ctdump` and `ctsave` commands.
- Lock tracing can optionally be enabled via another compilation option (-DLOCK_TRACE) to provide for tracing of locks, in order to find out the cause of some deadlocks. The `slstat` can then be used to see which locks were taken, by whom and who's waiting for these locks.
- Profiling can be enabled on selecive operations within the DDS Middleware via another conditional compilation option (-DPROFILE). The commands to control profiling are `profs` and `cprofs`.

We will not go into more detail here on these features, since they are quite advanced and require some deep insights into the working of Technicolor DDS, and are not intended for casual users.

List of Tables

THOMSON TELECOM BELGIUM NV
Prins Boudewijnlaan 47
2650 Edegem
Belgium

www.technicolor.com