OpenDDS

Release 3.16.0

Object Computing, Inc.

Jan 29, 2021

CONTENTS

Welcome to the documentation for OpenDDS 3.16.0!

CHAPTER

ONE

COMMON TERMS

1.1 Environment Variables

ACE_ROOT
: Root of the ACE source tree or installation prefix being used.

DDS_ROOT
: Root of the OpenDDS source tree or installation prefix being used.

TAO_ROOT
: Root of the TAO source tree or installation prefix being used.

CHAPTER

TWO

INTERNAL DOCUMENTATION

This documentation are for those who want to contribute to OpenDDS and those who are just curious!

2.1 OpenDDS Development Guidelines

This document organizes our current thoughts around development guidelines in a place that's readable and editable by the overall user and maintainer community. It's expected to evolve as different maintainers get a chance to review and contribute to it.

Although ideally all code in the repository would already follow these guidelines, in reality the code has evolved over many years by a diverse group of developers. At one point an automated re-formatter was run on the codebase, migrating from the GNU C style to the current, more conventional style, but automated tools can only cover a subset of the guidelines.

Table of Contents

        - *Order*
        - *Path*
    - *Time*

2.1.1 Repository

The repository is hosted on Github at objectcomputing/OpenDDS and is open for pull requests.

2.1.2 Automated Build Systems

Pull requests will be tested automatically and full CI builds of the master branch can be found at http://scoreboard.ociweb.com/oci-dds.html.

All tests listed in `DDS_ROOT/bin/dcps_tests.lst` are part of the automated test suite.

2.1.3 Doxygen

Doxygen is run on OpenDDS regularly. There are two hosted versions of this:

- Latest Release
  - Based on the current release of OpenDDS.
- Master
  - Based on the master branch in the repository. To access it, go to the scoreboard and click the green "Doxygen" link near the top.
  - Depending on the activity in the repository this might be unstable because of the time it takes to get the updated Doxygen on to the web sever. Prefer latest release unless working with newer code.

See *Documenting Code for Doxygen* to see how to take advantage of Doxygen when writing code in OpenDDS.

2.1.4 Dependencies

- MPC is the build system, used to configure the build and generate platform specific build files (Makefiles, VS solution files, etc.).
- ACE is a library used for cross-platform compatibility, especially networking and event loops. It is used both directly and through TAO.
- TAO is a C++ CORBA implementation built on ACE used extensively in the traditional OpenDDS operating mode which uses the DCPSInfoRepo. TAO types are also used in the End User DDS API. The TAO IDL compiler is used internally and by the end user to allow OpenDDS to use user defined IDL types as topic data.
- Perl is an interpreted language used in the configure script, the tests, and any other scripting in OpenDDS codebase.
- Google Test is required for OpenDDS tests. By default, CMake will be used to build a specific version of Google Test that we have as a submodule. An appropriate prebuilt or system Google Test can also be used.

See dependencies.md for all dependencies and details on how these are used in OpenDDS.

2.1.5 Text File Formatting

All text files in the source code repository follow a few basic rules. These apply to C++ source code, Perl scripts, MPC files, and any other plaintext file.

- A text file is a sequence of lines, each ending in the "end-of-line" character (AKA Unix line endings).
- Based on this rule, all files end with the end-of-line character.
- The character before end-of-line is a non-whitespace character (no trailing whitespace).
- Tabs are not used.
  - One exception, MPC files may contain literal text that's inserted into Makefiles which could require tabs.
  - In place of a tab, use a set number of spaces (depending on what type of file it is, C++ uses 2 spaces).
- Keep line length reasonable. I don't think it makes sense to strictly enforce an 80-column limit, but overly long lines are harder to read. Try to keep lines to roughly 80 characters.

2.1.6 C++ Standard

The C++ standard used in OpenDDS is C++03. There are some caveats to this but the OpenDDS must be able to be compiled with C++ 2003 compilers.

Use the C++ standard library as much as possible. The standard library should be preferred over ACE, which in turn should be preferred over system specific libraries. The C++ standard library includes the C standard library by reference, making those identifiers available in namespace std. Not all supported platforms have standard library support for wide characters (`wchar_t`) but this is rarely needed. Preprocessor macro `DDS_HAS_WCHAR` can be used to detect those platforms.

2.1.7 C++ Coding Style

- C++ code in OpenDDS must compile under the compilers listed in the `README.md` file.
- Commit code in the proper style from the start, so follow-on commits to adjust style don't clutter history.
- C++ source code is a plaintext file, so the guidelines in Text file formatting apply.
- A modified Stroustrup style is used (see tools/scripts/style).
  - Warning: not everything in tools/scripts/style represents the current guidelines.
- Sometimes the punctuation characters are given different names, this document will use:
  - Parentheses `( )`
  - Braces `{ }`
  - Brackets `[ ]`

Example

```
template<typename T>
class MyClass : public Base1, public Base2 {
public:
  bool method(const OtherClass& parameter, int idx = 0) const;
};

template<typename T>
bool MyClass<T>::method(const OtherClass& parameter, int) const
{
  if (parameter.foo() > 42) {
    return member_data_;
  } else {
    for (int i = 0; i < some_member_; ++i) {
      other_method(i);
    }
    return false;
  }
}
```

Punctuation

The punctuation placement rules can be summarized as:

- Open brace appears as the first non-whitespace character on the line to start function definitions.
- Otherwise the open brace shares the line with the preceding text.
- Parentheses used for control-flow keywords (`if`, `while`, `for`, `switch`) are separated from the keyword by a single space.
- Otherwise parentheses and brackets are not preceded by spaces.

Whitespace

- Each "tab stop" is two spaces.
- Namespace scopes that span most or all of a file do not cause indentation of their contents.
- Otherwise lines ending in `{` indicate that subsequent lines should be indented one more level until `}`.
- Continuation lines (when a statement spans more than one line) can either be indented one more level, or indented to nest "under" an `(` or similar punctuation.
- Add space around binary operators and after commas: `a + b`
- Do not add space around parentheses for function calls, a properly formatted function call looks like `func(arg1, arg2, arg3);`
- Do not add space around brackets for indexing, instead it should look like: `mymap[key]`
- In general, do not add space :) Do not add extra spaces to make syntax elements (that span lines/statements) line up. This only causes unnecessary changes in adjacent lines as the code evolves.

Language Usage

- Add braces following control-flow keywords even when they are optional.
- `this->` is not used unless required for disambiguation or to access members of a template-dependent base class.
- Declare local variables at the latest point possible.
- `const` is a powerful tool that enables the compiler to help the programmer find bugs. Use `const` everywhere possible, including local variables.
- Modifiers like `const` appear left of the types they modify, like: `const char* cstring = ...`. `char const*` is equivalent but not conventional.
- For function arguments that are not modified by the callee, pass by value for small objects (8 bytes?) and pass by const-reference for everything else.
- Arguments unused by the implementation have no names (in the definition that is, the declarations still have names), or a `/*commented-out*/` name.
- Use `explicit` constructors unless implicit conversions are intended and desirable.
- Use the constructor initializer list and make sure its order matches the declaration order.
- Prefer pre-increment/decrement (`++x`) to post-increment/decrement (`x++`) for both objects and non-objects.
- All currently supported compilers use the template inclusion mechanism. Thus function/method template definitions may not be placed in normal `*.cpp` files, instead they can go in `_T.cpp` (which are `#included` and not separately compiled), or directly in the `*.h`. In this case, `*_T.cpp` takes the place of `*.inl` (except it is always inlined). See ACE for a description of `*.inl` files.

Pointers and References

Pointers and references go along with the type, not the identifier. For example:

```
int* intPtr = &someInt;
```

Watch out for multiple declarations in one statement. `int* c, b;` does not declare two pointers! It's best just to break these into separate statements:

```
int* c;
int* b;
```

In code targeting C++03, `0` should be used as the null pointer. For C++11 and later, `nullptr` should be used instead. `NULL` should never be used.

Naming

(For library code that the user may link to)

- Preprocessor macros visible to user code must begin with `OPENDDS_`
- C++ identifiers are either in top-level namespace `DDS` (OMG spec defined) or `OpenDDS` (otherwise)
- Within the `OpenDDS` namespace there are some nested namespaces:
  - `DCPS`: anything relating to the implementation of the DCPS portion of the DDS spec
  - `RTPS`: types directly tied to the RTPS spec

  - `Federator`: DCPSInfoRepo federation
  - `FileSystemStorage`: reusable component for persistent storage
- Naming conventions
  - `ClassesAreCamelCaseWithInitialCapital`
  - `methodsAreCamelCaseWithInitialLower` OR `methods_are_lower_case_with_underscores`
  - `member_data_use_underscores_and_end_with_an_underscore_`
  - `ThisIsANamespaceScopedOrStaticClassMemberConstant`

Comments

- Add comments only when they will provide MORE information to the reader.
- Describing the code verbatim in comments doesn't add any additional information.
- If you start out implementation with comments describing what the code will do (or pseudocode), review all comments after implementation is done to make sure they are not redundant.
- Do not add a comment before the constructor that says `// Constructor`. We know it's a constructor. The same note applies to any redundant comment.

Documenting Code for Doxygen

Doxygen is run on the codebase with each change in master and each release. This is a simple guide showing the way of documenting in OpenDDS.

Doxygen supports multiple styles of documenting comments but this style should be used in non-trivial situations:

```
/**
 * This sentence is the brief description.
 *
 * Everything else is the details.
 */
class DoesStuff {
// ...
};
```

For simple things, a single line documenting comment can be made like:

```
/// Number of bugs in the code
unsigned bug_count = -1; // Woops
```

The extra `*` on the multiline comment and `/` on the single line comment are important. They inform Doxygen that comment is the documentation for the following declaration.

If referring to something that happens to be a namespace or other global object (like DDS, OpenDDS, or RTPS), you should precede it with a `%`. If not it will turn into a link to that object.

For more information, see the Doxygen manual.

Preprocessor

- If possible, use other language features things like inlining and constants instead of the preprocessor.
- Prefer `#ifdef` and `#ifndef` to `#if defined` and `#if !defined` when testing if a single macro is defined.
- Leave parentheses off preprocessor operators. For example, use `#if defined X && defined Y` instead of `#if defined(X) && defined(Y)`.
- As stated before, preprocessor macros visible to user code must begin with `OPENDDS_`.
- Ignoring the header guard if there is one, preprocessor statements should be indented using two spaces starting at the pound symbol, like so:

```
#if defined X && defined Y
#  if X > Y
#    define Z 1
#  else
#    define Z 0
#  endif
#else
#  define Z -1
#endif
```

Includes

Order

As a safeguard against headers being dependant on a particular order, includes should be ordered based on a hierarchy going from local headers to system headers, with spaces between groups of includes. This order can be generalized as the following:

1. The corresponding header to the source file (`Foo.h` if we were in `Foo.cpp`).
2. Headers from the local project.
3. Headers from external OpenDDS-based libraries.
4. Headers from `dds/DCPS`.
5. `dds/*C.h` Headers
6. Headers from external TAO-based libraries.
7. Headers from TAO.
8. Headers from external ACE-based libraries.
9. Headers from ACE.
10. Headers from external non-ACE-based libraries.
11. Headers from system and C++ standard libraries.

Path

Headers should only use local includes (`#include "foo/Foo.h"`) if the header is relative to the file. Otherwise system includes (`#include <foo/Foo.h>`) should be used to make it clear that the header is on the system include path.

In addition to this, includes for a file that will always be relative to the including file should have a relative include path. For example, a `dds/DCPS/bar.cpp` should include `dds/DCPS/bar.h` using `#include "bar.h"`, not `#include <dds/DCPS/bar.h>` and especially not `#include "dds/DCPS/bar.h"`.

Time

Measurements of time can be broken down into two basic classes: A specific point in time (Ex: 00:00 January 1, 1970) and a length or duration of time without context (Ex: 134 Seconds). In addition, a computer can change its clock while a program is running, which could mess up any time lapses being measured. To solve this problem, operating systems provide what's called a monotonic clock that runs independently of the normal system clock.

ACE can provide monotonic clock time and has a class for handling time measurements, `ACE_Time_Value`, but it doesn't differentiate between specific points in time and durations of time. It can differentiate between the system clock and the monotonic clock, but it does so poorly. OpenDDS provides three classes that wrap `ACE_Time_Value` to fill these roles: `TimeDuration`, `MonotonicTimePoint`, and `SystemTimePoint`. All three can be included using `dds/DCPS/TimeTypes.h`. Using `ACE_Time_Value` is discouraged unless directly dealing with ACE code which requires it and using `ACE_OS::gettimeofday()` or `ACE_Time_Value().now()` in C++ code in `dds/DCPS` treated as an error by the `lint.pl` linter script.

`MonotonicTimePoint` should be used when tracking time elapsed internally and when dealing with `ACE_Time_Value`s being given by the `ACE_Reactor` in OpenDDS. `ACE_Conditions`, like all ACE code, will default to using system time. Therefore the `Condition` class wraps it and makes it so it always uses monotonic time like it should. Like `ACE_OS::gettimeofday()`, referencing `ACE_Condition` in dds/DCPS will be treated as an error by `lint.pl`.

More information on using monotonic time with ACE can be found here.

`SystemTimePoint` should be used when dealing with the DDS API and timestamps on incoming and outgoing messages.

2.2 Bench 2 Performance & Scalability Test Framework

2.2.1 Motivation

The Bench 2 framework grew out of a desire to be able to test the performance and scalability of OpenDDS in large and heterogeneous deployments, along with the ability to quickly develop and deploy new test scenarios across a potentially-unspecified number of machines.

2.2.2 Overview

The resulting design of the Bench 2 framework depends on three primary test applications: worker processes, one or more node controllers, and a test controller.

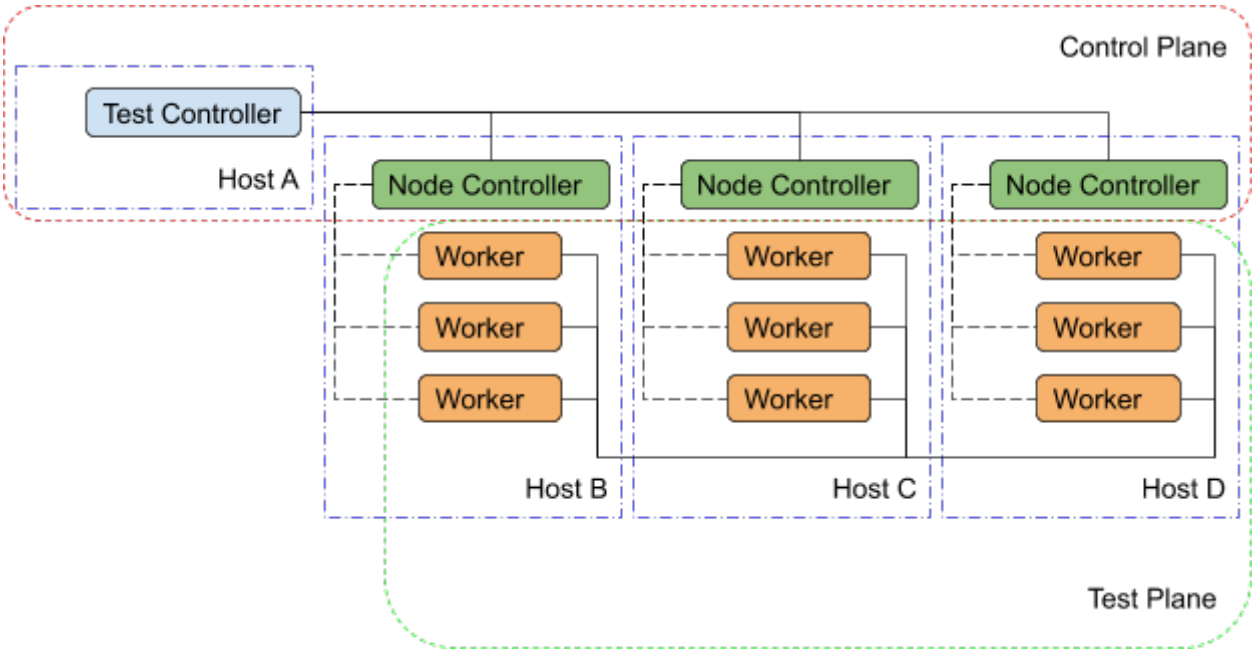

Fig. 2.1: Bench 2 Overview

Worker

The `worker` application, true to its name, performs most of the work associated with any given test scenario. It creates and exercises the DDS entities specified in its configuration file and gathers performance statistics related to discovery, data integrity, and performance. The worker's configuration file contains regions that may be used to represent OpenDDS's configuration sections as well as individual DDS entities and the QoS policies to be for their creation. In addition, the worker configuration contains test timing values and descriptions of test `actions` to be taken (e.g. publishing and forwarding data read from subscriptions). Upon test completion, the worker can write out a report file containing the performance statistics gathered during its run.

Node Controller

Each machine in the test environment will run (at least) one `node_controller` application which acts as a daemon and, upon request from a `test_controller`, will spawn one or more worker processes. Each request will contain the configuration to use for the spawned workers and, upon successful exit, the workers' report files will be read and sent back to the `test_controller` which requested it. Failed workers processes (aborts, crashes) will be noted and have their output logs sent back to the requesting `test_controller`. In addition to collecting worker reports, the node controller also gathers general system resource statistics during test execution (CPU and memory utilization) to be returned to the test controller at the end of the test.

Test Controller

Each execution of the test framework will use a `test_controller` to read in a scenario configuration file (an annotated collection of worker configuration file names) before listening for available `node_controller`'s and parceling out the scenario's worker configurations to the individual `node_controller`'s. The `test_controller` may also optionally adjust certain worker configuration values for the sake of the test (assigning a unique DDS partition to avoid collisions, coordinating worker test times, etc.). After sending the allocated scenario to each of the available node controllers, the test controller waits to receive reports from each of the node controllers. After receiving all the reports, the `test_controller` coalesces the performance statistics from each of the workers and presents the final results to the user (both on screen & in a results file).

2.2.3 Building Bench 2

Required Features

The primary requirements for building OpenDDS such that Bench 2 also gets built:

- C++11 Support (`--std=c++11`)
- RapidJSON present and enabled (`--rapidjson`)
- Tests are being built (`--tests`)

Required Targets

If these elements are present, you can either build the entire test suite (slow) or use these 3 targets (faster), which also cover all the required libraries:

- `Bench_Worker`
- `node_controller`
- `test_controller`

2.2.4 Running Bench 2

Environment Variables

To run Bench 2 executables with dynamically linked or shared libraries, you'll want to make sure the Bench 2 libraries are in your library path.

Linux/Unix

Add `${DDS_ROOT}/performance_tests/bench/lib` to your `LD_LIBRARY_PATH`

Windows

Add `%DDS_ROOT%\performance_tests\bench\lib` to your `PATH`

Assuming *`DDS_ROOT`* is already set on your system (from the `configure` script or from sourcing `setenv.sh`), there are convenience scripts to do this for you in the `bench` directory (`set_bench_env[.sh/.cmd]`)

Running a Bench 2 CI Test

In the event that you're debugging a failing Bench 2 CI test, you can use the run_test perl script found in the `bench` directory to execute the full scenario without first setting the environment as listed above, as the perl script sets it automatically before launching a single `node_controller` in the background and executing the test controller with the requested scenario. The perl script can be inspected in order to determine which scenarios have been made available in this way, and the script can be modified to easily run other scenarios against a single node controller with relative ease.

Running Scenarios Manually

Assuming you already have scenario and worker configuration files defined, the general approach to running a scenario is to start one or more `node_controllers` (across one or more hosts) and then execute the test_controller with the desired scenario configuration.

2.2.5 Configuration Files

As a rule, Bench 2 uses JSON configuration files that directly map onto the C++ Platform Specific Model (PSM) of the IDL found in `bench/idl` and the IDL used in the DDS specification. This allows the test applications to easily convert between configuration files and C++ structures useful for the configuration of DDS entities.

Scenario Configuration Files

Scenario configuration files are used by the test controller to determine the number and type (configuration) of worker processes required for a particular test scenario. In addition, the scenario file may specify certain sets of workers to be run on the same node by placing them together in a node "prototype" (see below).

IDL Definition

```
struct WorkerPrototype {
  // Filename of the JSON Serialized Bench::WorkerConfig
  string config;
  // Number of workers to spawn using this prototype (Must be >=1)
  unsigned long count;
};

typedef sequence<WorkerPrototype> WorkerPrototypes;

struct NodePrototype {
  // Assign to a node controller with a name that matches this wildcard
  string name_wildcard;
  WorkerPrototypes workers;
  // Number of Nodes to spawn using this prototype (Must be >=1)
```

(continues on next page)

(continued from previous page)

```
  unsigned long count;
  // This NodePrototype must have a Node to itself
  boolean exclusive;
};

typedef sequence<NodePrototype> NodePrototypes;

// This is the root type of the scenario configuration file
struct ScenarioPrototype {
  string name;
  string desc;
  // Workers that must be deployed in sets
  NodePrototypes nodes;
  // Workers that can be assigned to any node
  WorkerPrototypes any_node;
  /*
   * Number of seconds to wait for the scenario to end.
   * 0 means never timeout.
   */
  unsigned long timeout;
};
```

Annotated Example

```
{
  "name": "An Example",
  "desc": "This shows the structure of the scenario configuration",
  "nodes": [
    {
      "name_wildcard": "example_nc_*",
      "workers": [
        {
          "config": "daemon.json",
          "count": 1
        },
        {
          "config": "spawn.json",
          "count": 1
        }
      ],
      "count": 2,
      "exclusive": false
    }
  ],
  "any_node": [
    {
      "config": "master.json",
      "count": 1
    }
  ],
  "timeout": 120
}
```

This scenario configuration will launch 5 worker processes. It will launch 2 pairs of “daemon” / “spawn” processes, with each member of each pair being kept together on the same node (i.e. same `node_controller`). The pairs

themselves may be split across nodes, but each "daemon" will be with at least one "spawn" and vice-versa. They may also wind up all together on the same node, depending on the number of available nodes. And finally, one "master" process will be started wherever there is room available.

The "name_wildcard" field is used to filter the `node_controller` instances that can be used to host the nodes in the current node config - only the `node_controller` instances with names matching the wildcard can be used. If the "name_wildcard" is omitted or its value is empty, any `node_controller` can be used. If node "prototypes" are marked exclusive, the test controller will attempt to allocate them exclusively to their own node controllers. If not enough node controllers exist to honor all the exclusive nodes, the test controller will fail with an error message.

Worker Configuration Files

QoS Masking

In a typical DDS application, default QoS objects are often supplied by the entity factory so that the application developer can make required changes locally and not impact larger system configuration choices. As such, the QoS objects found within the JSON configuration file should be treated as a "delta" from the default configuration object of a parent factory class. So while the JSON "qos" element names will directly match the relevant IDL element names, there will also be an additional "qos_mask" element that lives alongside the "qos" element in order to specify which elements apply. For each QoS attribute "attribute" within the "qos" object, there will also be a boolean "has_attribute" within the "qos_mask" which informs the builder library that this attribute should indeed be applied against the default QoS object supplied by the parent factory class before the entity is created.

IDL Definition

```
struct TimeStamp {
  long sec;
  unsigned long nsec;
};

typedef sequence<string> StringSeq;
typedef sequence<double> DoubleSeq;

enum PropertyValueKind { PVK_TIME, PVK_STRING, PVK_STRING_SEQ, PVK_STRING_SEQ_SEQ,␣
↪PVK_DOUBLE, PVK_DOUBLE_SEQ, PVK_ULL };
union PropertyValue switch (PropertyValueKind) {
  case PVK_TIME:
    TimeStamp time_prop;
  case PVK_STRING:
    string string_prop;
  case PVK_STRING_SEQ:
    StringSeq string_seq_prop;
  case PVK_STRING_SEQ_SEQ:
    StringSeqSeq string_seq_seq_prop;
  case PVK_DOUBLE:
    double double_prop;
  case PVK_DOUBLE_SEQ:
     DoubleSeq double_seq_prop;
  case PVK_ULL:
    unsigned long long ull_prop;
};

struct Property {
  string name;
  PropertyValue value;
};
```

(continues on next page)

(continued from previous page)

```
typedef sequence<Property> PropertySeq;

struct ConfigProperty {
  string name;
  string value;
};
typedef sequence<ConfigProperty> ConfigPropertySeq;

// ConfigSection

struct ConfigSection {
  string name;
  ConfigPropertySeq properties;
};
typedef sequence<ConfigSection> ConfigSectionSeq;

// Writer

struct DataWriterConfig {
  string name;
  string topic_name;
  string listener_type_name;
  unsigned long listener_status_mask;
  string transport_config_name;
  DDS::DataWriterQos qos;
  DataWriterQosMask qos_mask;
};
typedef sequence<DataWriterConfig> DataWriterConfigSeq;

// Reader

struct DataReaderConfig {
  string name;
  string topic_name;
  string listener_type_name;
  unsigned long listener_status_mask;
  PropertySeq listener_properties;
  string transport_config_name;
  DDS::DataReaderQos qos;
  DataReaderQosMask qos_mask;
  StringSeq tags;
};
typedef sequence<DataReaderConfig> DataReaderConfigSeq;

// Publisher

struct PublisherConfig {
  string name;
  string listener_type_name;
  unsigned long listener_status_mask;
  string transport_config_name;
  DDS::PublisherQos qos;
  PublisherQosMask qos_mask;
  DataWriterConfigSeq datawriters;
};
typedef sequence<PublisherConfig> PublisherConfigSeq;
```

(continues on next page)

(continued from previous page)

```
// Subscription

struct SubscriberConfig {
  string name;
  string listener_type_name;
  unsigned long listener_status_mask;
  string transport_config_name;
  DDS::SubscriberQos qos;
  SubscriberQosMask qos_mask;
  DataReaderConfigSeq datareaders;
};
typedef sequence<SubscriberConfig> SubscriberConfigSeq;

// Topic

struct ContentFilteredTopic {
  string cft_name;
  string cft_expression;
  DDS::StringSeq cft_parameters;
};

typedef sequence<ContentFilteredTopic> ContentFilteredTopicSeq;

struct TopicConfig {
  string name;
  string type_name;
  DDS::TopicQos qos;
  TopicQosMask qos_mask;
  string listener_type_name;
  unsigned long listener_status_mask;
  string transport_config_name;
  ContentFilteredTopicSeq content_filtered_topics;
};
typedef sequence<TopicConfig> TopicConfigSeq;

// Participant

struct ParticipantConfig {
  string name;
  unsigned short domain;
  DDS::DomainParticipantQos qos;
  DomainParticipantQosMask qos_mask;
  string listener_type_name;
  unsigned long listener_status_mask;
  string transport_config_name;
  StringSeq type_names;
  TopicConfigSeq topics;
  PublisherConfigSeq publishers;
  SubscriberConfigSeq subscribers;
};
typedef sequence<ParticipantConfig> ParticipantConfigSeq;

// TransportInstance

struct TransportInstanceConfig {
  string name;
  string type;
```

(continues on next page)

(continued from previous page)

```
  unsigned short domain;
};
typedef sequence<TransportInstanceConfig> TransportInstanceConfigSeq;

// Discovery

struct DiscoveryConfig {
  string name;
  string type; // "rtps" or "repo"
  string ior; // "repo" URI (e.g. "file://repo.ior")
  unsigned short domain;
};
typedef sequence<DiscoveryConfig> DiscoveryConfigSeq;

// Process

struct ProcessConfig {
  ConfigSectionSeq config_sections;
  DiscoveryConfigSeq discoveries;
  TransportInstanceConfigSeq instances;
  ParticipantConfigSeq participants;
};

// Worker

// This is the root structure of the worker configuration
// For the sake of readability, module names have been omitted
// All structures other than this one belong to the Builder module
struct WorkerConfig {
  TimeStamp create_time;
  TimeStamp enable_time;
  TimeStamp start_time;
  TimeStamp stop_time;
  TimeStamp destruction_time;
  PropertySeq properties;
  ProcessConfig process;
  ActionConfigSeq actions;
  ActionReportSeq action_reports;
};
```

Annotated Example

```
{
  "create_time": { "sec": -1, "nsec": 0 },
```

Since the timestamp is negative, this treats the time as relative and waits one second.

```
"enable_time": { "sec": -1, "nsec": 0 },
"start_time": { "sec": 0, "nsec": 0 },
```

Since the time is zero and thus neither absolute nor relative, this treats the time as indefinite and waits for keyboard input from the user.

```
"stop_time": { "sec": -10, "nsec": 0 },
```

Again, a relative timestamp. This time, it waits for 10 seconds for the test actions to run before stopping the test.

```
"destruction_time": { "sec": -1, "nsec": 0 },

"process": {
```

This is the primary section where all the DDS entities are described, along with configuration of OpenDDS.

```
"config_sections": [
```

The elements of this section are functionally identical to the sections of an OpenDDS `.ini` file with the same name. Each config section is created programmatically within the worker process using the name provided and made available to the OpenDDS `ServiceParticipant` during entity creation. The example here sets the value of both the `DCPSSecurity` and `DCPSDebugLevel` keys to 0 within the `[common]` section of the configuration.

```
  { "name": "common",
    "properties": [
      { "name": "DCPSSecurity",
        "value": "0"
      },
      { "name": "DCPSDebugLevel",
        "value": "0"
      }
    ]
  }
],
"discoveries": [
```

Even if there is no configuration section for it (see above), this allows us to create unique discovery instances per domain. If both are specified, this will find and use / modify the one specified in the configuration section above. Valid types are `"rtps"` and `"repo"` (requires additional `"ior"` element with valid URL)

```
  { "name": "bench_test_rtps",
    "type": "rtps",
    "domain": 7
  }
],
"instances": [
```

Even if there is no configuration section for it (see above), this allows us to create unique transport instances. If both are specified, this will find and use / modify the one specified in the configuration section above. Valid types are `rtps_udp`, `tcp`, `udp`, `ip_multicast`, `shmem`.

```
  { "name": "rtps_instance_01",
    "type": "rtps_udp",
    "domain": 7
  }
],
"participants": [
```

The list of participants to create.

```
{ "name": "participant_01",
  "domain": 7,
  "transport_config_name": "rtps_instance_01",
```

The transport config that gets bound to this participant

```
"qos": { "entity_factory": { "autoenable_created_entities": false } },
"qos_mask": { "entity_factory": { "has_autoenable_created_entities": false } },
```

An example of QoS masking. Note that in this example, the boolean flag is false, so the QoS mask is not actually applied. In this case, both lines here were added to make switching back and forth between `autoenable_created_entities` easier (simply change the value of the bottom element `"has_autoenable_created_entities"` to `"true"`).

```
"topics": [
```

List of topics to register for this participant

```
{ "name": "topic_01",
  "type_name": "Bench::Data"
```

Note the type name. `"Bench::Data"` is currently the only topic data type supported by the Bench 2 framework. That said, it contains a variably sized array of octets, allowing a configurable range of data payload sizes (see write_action below).

```
"content_filtered_topics": [
  {
    "cft_name": "cft_1",
    "cft_expression": "filter_class > %0",
    "cft_parameters": ["2"]
  }
]
```

List of content filtered topics. Note “cft_name”. Its value can be used in DataReader “topic_name” to use the content filter.

```
   }
 ],
"subscribers": [
```

List of subscribers

```
{ "name": "subscriber_01",
  "datareaders": [
```

List of DataReaders

```
{ "name": "datareader_01",
  "topic_name": "topic_01",
  "listener_type_name": "bench_drl",
  "listener_status_mask": 4294967295,
```

Note the listener type and status mask. `"bench_drl"` is a listener type registered by the Bench Worker application that does most of the heavy lifting in terms of stats calculation and reporting. The mask is a fully-enabled bitmask for all listener events (i.e. `2^32 - 1`).

```
"qos": { "reliability": { "kind": "RELIABLE_RELIABILITY_QOS" } },
"qos_mask": { "reliability": { "has_kind": true } },
```

DataReaders default to best effort QoS, so here we are setting the reader to reliable QoS and flagging the `qos_mask` appropriately in order to get a reliable datareader.

```
"tags": [ "my_topic", "reliable_transport" ]
```

The config can specify a list of tags associated with each data reader. The statistics for each tag is computed in addition to the overall statistics and can be printed out at the end of the run by the `test_controller`.

```
      }
    ]
  }
],
"publishers": [
```

List of publishers within this participant

```
{ "name": "publisher_01",
  "datawriters": [
```

List of DataWriters within this publisher

```
{ "name": "datawriter_01",
```

Note that each DDS entity is given a process-entity-unique name, which can be used below to locate / identify this entity.

```
              "topic_name": "topic_01",
              "listener_type_name": "bench_dwl",
              "listener_status_mask": 4294967295
            }
          ]
        }
      ]
    }
  ]
},
"actions": [
```

A list of worker 'actions' to start once the test 'start' period begins.

```
{
  "name": "write_action_01",
  "type": "write",
```

Current valid types are "write", "forward", and "set_cft_parameters"

```
"writers": [ "datawriter_01" ],
```

Note the datawriter name defined above is passed into the action's writer list. This is used to locate the writer within the process.

```
"params": [
  { "name": "data_buffer_bytes",
```

The size of the octet array within the `Bench::Data` message. Note, actual messages will be slightly larger than this value.

```
  "value": { "_d": "PVK_ULL", "ull_prop": 512 }
},
{ "name": "write_frequency",
```

The frequency with which the write action attempts to write a message. In this case, twice a second.

```
  "value": { "_d": "PVK_DOUBLE", "double_prop": 2.0 }
},
```

```
{ "name": "filter_class_start_value",
  "value": { "_d": "PVK_ULL", "ull_prop": 0 }
},
{ "name": "filter_class_stop_value",
  "value": { "_d": "PVK_ULL", "ull_prop": 0 }
},
{ "name": "filter_class_increment",
  "value": { "_d": "PVK_ULL", "ull_prop": 0 }
}
```

Value range and increment for "filter_class" data variable, used when writing data. This variable is an unsigned integer intended to be used for content filtered topics "set_cft_parameters" actions

```
  ]
},

{ "name": "cft_action_01",
  "type": "set_cft_parameters",
  "params": [
    { "name": "content_filtered_topic_name",
      "value": { "_d": "PVK_STRING", "string_prop": "cft_1" }
    },
    { "name": "max_count",
      "value": { "_d": "PVK_ULL", "ull_prop": 3 }
    },
```

Maximum count of "Set" actions to be taken

```
{ "name": "param_count",
  "value": { "_d": "PVK_ULL", "ull_prop": 1 }
},
```

Number of parameters to be set

```
{ "name": "set_frequency",
  "value": { "_d": "PVK_DOUBLE", "double_prop": 2.0 }
},
```

The frequency for set action, per second

```
{ "name": "acceptable_param_values",
  "value": { "_d": "PVK_STRING_SEQ_SEQ", "string_seq_seq_prop": [ ["1", "2", "3"] ] }
},
```

Lists of allowed values to set to, for each parameter. Worker will iterate throught the list sequentially unless "random_order" flag (below) is specified

```
        { "name": "random_order",
          "value": { "_d": "PVK_ULL", "ull_prop": 1 }
        }
      ]
  }
```

(continues on next page)

(continued from previous page)

```
  ]
}
```

2.2.6 Detailed Application Descriptions

test_controller

As mentioned above, the `test_controller` application is the application responsible for running test scenarios and, as such, will probably wind up being the application most frequently run directly by testers. The `test_controller` needs network visibility to at least one `node_controller` configured to run on the same domain. It expects, as arguments, the path to a directory containing config files (both scenario & worker) and the name of a scenario configuration file to run (without the `.json` extension). For historical reasons, the config directory is often simply called "example". The `test_controller` application also supports a number of optional configuration parameters, some of which are described in the section below.

Usage

```
test_controller CONFIG_PATH SCENARIO_NAME [OPTIONS]
```

```
test_controller --help|-h
```

This is a subset of the options. Use `--help` option to see all the options.

CONFIG_PATH
: Path to the directory of the test configurations and artifacts

SCENARIO_NAME
: Name of the scenario file in the test context without the *.json* extension.

--domain `N`
: The DDS Domain to use. The default is 89.

--wait-for-nodes `N`
: The number of seconds to wait for nodes before broadcasting the scenario to them. The default is 10 seconds.

--timeout `N`
: The number of seconds to wait for a scenario to complete. Overrides the value defined in the scenario. If N is 0, there is no timeout.

--override-create-time `N`
: Overwrite individual worker configs to create their DDS entities N seconds from now (absolute time reference)

--override-start-time `N`
: Overwrite individual worker configs to start their test actions (writes & forwards) N seconds from now (absolute time reference)

--tag `TAG`
: Specify a tag for which the performance statistics will be printed out (and saved to a results file). Multiple instances of this option can be specified, each for a single tag.

--json-result-id `ID`
: Specify a name to store the raw JSON report under. By default, this not enabled. These results will contain the full raw Bench::TestController report, including all node controller and worker reports (and DDS entity reports)

node_controller

The node controller application is best thought of as a daemon, though the application can be run both in a long-running `daemon` mode and also a `one-shot` mode more appropriate for testing. The `daemon-exit-on-error` mode additionally has the ability to exit the process every time an error is encountered, which is useful for restarting the application when errors are detected, if run as a part of an OS system environment (systemd, supervisord, etc).

Usage

```
node_controller [OPTIONS] one-shot|daemon|daemon-exit-on-error
```

one-shot
: Run a single batch of worker requests (configs > processes > reports) and report the results before exiting. Useful for one-off and local testing.

daemon
: Act as a long-running process that continually runs batches of worker requests, reporting the results. Attempts to recover from errors.

daemon-exit-on-error
: Act as a long-running process that continually runs batches of worker requests, reporting the results. Does not attempt to recover from errors.

--domain `N`
: The DDS Domain to use. The default is 89.

--name `STRING`
: Human friendly name for the node. Will be used by the test controller for referring to the node. During allocation of node controllers, the name is used to match against the "name_wildcard" fields of the node configs. Only node controllers whose names match the "name_wildcard" of a given node config can be allocated to that node config. Multiple nodes could have the same name.

worker

The worker application is meant to mimic the behavior of a single arbitrary OpenDDS test application. It uses the Bench builder library along with its JSON configuration file to first configure OpenDDS (including discovery & transports) and then create all required DDS entities using any desired DDS QoS attributes. Additionally, it allows the user to configure several test phase timing parameters, using either absolute or relative times:

- DDS entity creation (`create_time`)
- DDS entity "enabling" (`enable_time`) (only relevant if `autoenable_created_entities` QoS setting is false)
- test actions start time (`start_time`)
- test actions stop time (`stop_time`)
- DDS entity destruction (`destruction_time`)

Finally, it also allows for the configuration and execution of test "actions" which take place between the "start" and "stop" times indicated in configuration. These may make use of the created DDS entities in order to simulate application behavior. At the time of this writing, the three actions are "write", which will write to a datawriter using data of a configurable size and frequency (and maximum count), "forward", which will pass along the data read from one datareader to a datawriter, allowing for more complex test behaviors (including round-trip latency & jitter calculations), and "set_cft_parameters", which will change the content filtered topic parameter values dynamically. In addition to

reading a JSON configuration file, the worker is capable of writing a JSON report file that contains various test statistics gathered from listeners attached to the created DDS entities. This report is read by the `node_controller` after the worker process ends and is then sent back to the waiting `test_controller`.

Usage

```
worker [OPTIONS] CONFIG_FILE
```

--log LOG_FILE
: The log file path. Will log to *stdout* if not passed.

--report REPORT_FILE
: The report file path.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

C

D

E

N

O

S

T

W