OpenDDS

Release 3.19.0

Object Computing, Inc.

Dec 11, 2021

CONTENTS

Welcome to the documentation for OpenDDS 3.19.0!

It is available for download on GitHub.

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.

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.

See *Running Tests* for how tests are run in general. See *GitHub Actions Summary and FAQ* for how building and testing is done with GitHub Actions.

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 docs/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. Generated headers from the same directory should be placed last within these groups. This order can be generalized as the following:

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

There can be exceptions to this list. For example if a header from ACE or the system library was needed to decide if another header should be included.

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"`.

Example

For a `Doodad.cpp` file in dds/DCPS, the includes could look like:

```
#include <DCPS/DdsDcps_pch.h>

#include "Doodad.h"

#include <ace/config-lite.h>
#ifndef ACE_CPP11
#  include "ConditionVariable.h"
#endif
#include "ReactorTask.h"
#include "transport/framework/DataLink.h"
```

(continues on next page)

(continued from previous page)

```
#include <dds/DdsDcpsCoreC.h>

#include <tao/Version.h>

#include <ace/Version.h>

#include <openssl/opensslv.h>

#include <unistd.h>
#include <stdlib.h>
```

2.1.8 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_Values` 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.

2.1.9 Logging

ACE Logging

Logging is done via ACE's logging macro functions, `ACE_DEBUG` and `ACE_ERROR`, defined in `ace/Log_Msg.h`. The logging macros arguments to both are:

- A `ACE_Log_Priority` value
  - This is an enum defined in `ace/Log_Priority.h` to say what the priority or severity of the message is.
- The format string
  - This is similar to the format string for the standard `printf`, where it substitutes sequences starting with `%`, but the format of theses sequences is different. For example `char*` values are substituted using `%C` instead

of `%s`. See the documenting comment for `ACE_Log_Msg::log` in `ace/Log_Msg.h` for what the format of the string is.

- The variable number of arguments
  - Like `printf` the variable arguments can't be whole objects, like a `std::string` value. In the case of `std::string`, the format and arguments would look like: `"%C", a_string.c_str()`.

Note that all the `ACE_DEBUG` and `ACE_ERROR` arguments must be surrounded by two sets of parentheses.

```
ACE_DEBUG((LM_DEBUG, "Hello, %C!\n", "world"));
```

ACE logs to `stderr` by default on conventional platforms, but can log to other places.

Usage in OpenDDS

Logging Conditions and Priority

In OpenDDS `ACE_DEBUG` and `ACE_ERROR` are used directly most of the time, but sometimes they are used indirectly, like with the transport framework's `VDBG` and `VDBG_LVL`. They also should be conditional on one of the logging control systems in OpenDDS. See section 7.6 of the OpenDDS Developer's Guide for user perspective.

The logging conditions are as follows:

Message Kind	Macro	Priority	Condition
Unrecoverable error	`ACE_ERROR`	`LM_ERROR`	`log_level >= LogLevel::Error`
Unreportable recoverable error	`ACE_ERROR`	`LM_WARNING`	`log_level >= LogLevel::Warning`
Reportable recoverable error	`ACE_ERROR`	`LM_NOTICE`	`log_level >= LogLevel::Notice`
Informational message	`ACE_DEBUG`	`LM_INFO`	`log_level >= LogLevel::Info`
Debug message	`ACE_DEBUG`	`LM_DEBUG`	Based on `DCPS_debug_level` or one of the other debug systems *listed below*[1]

An *unrecoverable error* indicates that OpenDDS is in a state where it cannot function as intended. This may be the result of a defect, misconfiguration, or interference.

A *recoverable error* indicates that OpenDDS could not perform a desired action but remains in a state where it can function as intended.

A *reportable error* indicates that OpenDDS can report the error via the API through something like an exception or return value.

An *informational message* gives high level information mostly at startup, like the version of OpenDDS being used.

A *debug message* gives lower level information, such as if a message is being sent. These are directly controlled by one of a few debug logging control systems.

- `DCPS_debug_level` should be used for all debug logging that doesn't fall under the other systems. It is an unsigned integer value which ranges from 0 to 10. See dds/DCPS/debug.h for details.
- `Transport_debug_level` should be used in the transport layer. It is an unsigned integer value which ranges from 0 to 6. See dds/DCPS/transport/framework/TransportDebug.h for details.
- `security_debug` should be used for logging in related to DDS Security. It is an object with `bool` members that make up categories of logging messages that allow fine control. See dds/DCPS/debug.h for details.

[1] Debug messages don't rely on both *LogLevel::Debug* and a debug control system. The reason is that it results in a simpler check and the log level already loosely controls all the debug control systems. See the *LogLevel::set* function in dds/DCPS/debug.cpp for exactly what it does.

For number-based conditions like `DCPS_debug_level` and `Transport_debug_level`, the number used should be the log level the message starts to become active at. For example for `DCPS_debug_level >= 6` should be used instead of `DCPS_debug_level > 5`.

Message Content

- Log messages should take the form:

```
(%P|%t) [ERROR:|WARNING:|NOTICE:] FUNCTION_NAME: MESSAGE\n
```

  - Use `ERROR:`, `WARNING:`, and `NOTICE:` if using the corresponding log priorities.
  - `CLASS_NAME::METHOD_NAME` should be used instead of just the function name if it's part of a class. It's at the developer's discretion to come up with a meaningful name for members of overload sets, templates, and other more complex cases.
  - `security_debug` and `transport_debug` log messages should indicate the category name, for example:

```
if (security_debug.access_error) {
  ACE_ERROR((LM_ERROR, "(%P|%t) ERROR: {access_error} example_function: Hello,␣
↪World!\n"));
}
```

- Format strings should not be wrapped in `ACE_TEXT`. We shouldn't go out of our way to replace it in existing logging points, but it should be avoided it in new ones.
  - `ACE_TEXT`'s purpose is to wrap strings and characters in `L` on builds where `uses_wchar=1`, so they become the wide versions.
  - While not doing it might result in a performance hit for character encoding conversion at runtime, the builds where this happens are rare, so the it's outweighed by the added visual noise to the code and the possibility of bugs introduced by improper use of `ACE_TEXT`.
- Avoid new usage of `ACE_ERROR_RETURN` in order to not hide the return statement within a macro.

Examples

```
if (log_level >= LogLevel::Error) {
  ACE_ERROR((LM_DEBUG, "(%P|%t) ERROR: example_function: Hello, World!\n"));
}

if (log_level >= LogLevel::Warning) {
  ACE_ERROR((LM_WARNING, "(%P|%t) WARNING: example_function: Hello, World!\n"));
}

if (log_level >= LogLevel::Notice) {
  ACE_ERROR((LM_NOTICE, "(%P|%t) NOTICE: example_function: Hello, World!\n"));
}

if (log_level >= LogLevel::Info) {
  ACE_DEBUG((LM_INFO, "(%P|%t) example_function: Hello, World!\n"));
}

if (DCPS_debug_level >= 1) {
```

(continues on next page)

(continued from previous page)

```
  ACE_DEBUG((LM_DEBUG, "(%P|%t) example_function: Hello, World!\n"));
}
```

2.2 Documentation Guidelines

This Sphinx-based documentation is hosted on Read the Docs and can be located here. It can also be built locally. To do this follow the steps in the following section.

2.2.1 Building

Run docs/build.py, passing the kinds of documentation desired. Multiple kinds can be passed, and they are documented in the following sections.

Requirements

The script requires Python 3.6 or later and an internet connection if the script needs to download dependencies or check the validity of external links.

You might receive a message like this when running for the first time:

```
build.py: Creating venv...
The virtual environment was not created successfully because ensurepip is not
available.  On Debian/Ubuntu systems, you need to install the python3-venv
package using the following command.

    apt install python3.9-venv
```

If you do, then follow the directions it gives, remove the `docs/.venv` directory, and try again.

HTML

HTML documentation can be built and viewed using `docs/build.py -o html`. If it was built successfully, then the front page will be at `docs/_build/html/index.html`.

PDF

Note: This has additional dependencies on LaTeX that are documented here.

PDF documentation can be built and viewed using `docs/build.py -o pdf`. If it was built successfully, then the PDF file will be at `docs/_build/latex/opendds.pdf`.

Dash

Documentation can be built for Dash, Zeal, and other Dash-compatible applications using doc2dash. The command for this is `docs/build.py dash`. This will create a `docs/_build/OpenDDS.docset` directory that must be manually moved to where other docsets are stored.

Strict Checks

`docs/build.py strict` will promote Sphinx warnings to errors and check to see if links resolve to a valid web page.

Note: The documenation includes dynamic links to files in the GitHub repo created by *ghfile*. These links will be invalid until the git commit they were built under is pushed to a Github fork of OpenDDS. This also means running will cause those links to marked as broken. A workaround for this is to pass `-c master` or another commit, branch, or tag that is desired.

Building Manually

It is recommended to use `build.py` to build the documentation as it will handle dependencies automatically. If necessary though, Sphinx can be ran directly.

To build the documentation the dependencies need to be installed first. Run this from the `docs` directory to do this:

```
pip3 install -r requirements.txt
```

Then `sphinx-build` can be ran. For example to build the HTML documentation:

```
sphinx-build -M html . _build
```

2.2.2 RST/Sphinx Usage

- See Sphinx reStructuredText Primer for basic RST usage.
- Inline code such as class names like `DataReader` and other symbolic text such as commands like `ls` should use double backticks: ``` ``TEXT`` ```. This distinguishes it as code, makes it easier to distinguish characters, and reduces the chance of needing to escape characters if they happen to be special for RST.
- One sentence per line should be perfered. This makes it easier to see what changed in a `git diff` or GitHub PR and easier to move sentences around in editors like Vim. It also avoids inconsistencies involving what the maximum line length is. This might make it more annoying to read the documentation raw, but that's not the indented way to do so anyway.

GitHub Links

There are a few shortcuts for linking to the GitHub repository that are custom to OpenDDS. These come of the form of RST roles and are implemented in docs/sphinx_extensions/github_links.py.

ghfile

```
:ghfile:`README.md`

:ghfile:`the \`\`README.md\`\` File <README.md>`

:ghfile:`the support section of the \`\`README.md\`\` File <README.md#support>`

:ghfile:`check out the available support <README.md#support>`
```

Turns into:

README.md#support

README.md

the `README.md` File

the support section of the `README.md` File

check out the available support

The path passed must exist, be relative to the root of the repository, and will have to be committed, if it's not already. If there is a URL fragment in the path, like `README.md#support`, then it will appear in the link URL.

It will try to point to the most specific version of the file:

- If `-c` or `--gh-links-commit` was passed to `build.py`, then it will use the commit, branch, or tag that was passed along with it.
- Else if the OpenDDS is a release it will calculate the release tag and use that.
- Else if the OpenDDS is in a git repository it will use the commit hash.
- Else it will use `master`.

ghissue

```
:ghissue:`213`

:ghissue:`this is the issue <213>`

:ghissue:`this is **the issue** <213>`
```

Turns into:

Issue #213 on GitHub

this is the issue

this is **the issue**

ghpr

```
:ghpr:`1`

:ghpr:`this is the PR <1>`

:ghpr:`this is **the PR** <1>`
```

Turns into:

Pull Request #1 on GitHub

this is the PR

this is **the PR**

2.3 Unit Tests

2.3.1 The Goals of Unit Testing

The primary goal of a unit test is to provide informal evidence that a piece of code performs correctly. An alternative to unit testing is writing formal proofs. However, formal proofs are difficult, expensive, and unmaintainable given the changing nature of software. Unit tests, while necessarily incomplete, are a practical alternative.

Unit tests document how to use various algorithms and data structures and serve as an informal set of requirements. As such, a unit test should be developed with the idea that it will serve as a reference for future developers. Clarity in unit tests serve to accomplish their primary goal of establishing correctness. That is, a unit test that is difficult to understand casts doubt that the code being tested is correct. Consequently, unit tests should be clear and concise.

The confidence one has in a piece of code is often related to the number of code paths explored in it. This is often approximated by "code coverage." That is, one can run the unit test with a coverage tool to see which code paths were exercised by the unit test. Code with higher coverage tends to have fewer bugs because the tester has often considered various corner-cases. Consequently, unit tests should aim for high code coverage.

Unit tests should be executed frequently to provide developers with instant feedback. This applies to the feature under development and the system as a whole. That is, developers should frequently execute all of the unit tests to make sure they haven't broken functionality elsewhere in the system. The more frequently the tests are run, the smaller the increment of development and the easier it is to identify a breaking change. Thus, unit tests should execute quickly.

Code that is difficult to test will most likely be difficult to use. Code that is difficult to use correctly will lead to bugs in code that uses it. Consequently, unit tests are vital to the design of useful software as developing a unit test provides feedback on the design of the code under test. Often, when developing a unit test, one will find parts of the design that can be improved.

Unit tests should promote and not inhibit development. A robust set of unit tests allows a developer to aggressively refactor since the correctness of the system can be checked after the refactoring. However, unit tests do produce drag on development since they must be maintained as the code evolves. Thus, it is important that the unit test code be properly maintained so that they are an asset and not a liability.

Some of the goals mentioned above are in conflict. Adding code to increase coverage may make the tests less maintainable, slower, and more difficult to understand. The following metrics can be generated to measure the utility of the unit tests:

- Code coverage
- Test compilation time

- Test execution time
- Test code size vs. code size
- Defect rate vs. code coverage (Are bugs appearing in code that is not tested as well?)

2.3.2 Unit Test Organization

The most basic unit when testing is the *test case*. A test case typically has four phases.

1. Setup - The system is initialized to a known state.
2. Exercise - The code under test is invoked.
3. Check - The resulting state of the system and outputs are checked.
4. Teardown - Any resources allocated in the test are deallocated.

Test cases are grouped into a *test suite*.

Test suites are organized into *a test plan*.

We adopt file boundaries for organizing the unit tests for OpenDDS. That is, the unit tests for a file group `dds/DCPS/SomeFile.(h|cpp)` will be located in `tests/unit-tests/dds/DCPS/SomeFile.cpp`. The file `tests/unit-tests/dds/DCPS/SomeFile.cpp` is a test suite containing all of the test cases for `dds/DCPS/SomeFile.(h|cpp)`. The test plan for OpenDDS will execute all of the test suites under tests/unit-tests. When the complete test plan takes too much time to execute, it will be sub-divided along module boundaries.

In regards to coverage, the coverage of `dds/DCPS/SomeFile.(h|cpp)` is measured by executing the tests in its test suite `tests/unit-tests/dds/DCPS/SomeFile.cpp`. The purpose of this is to avoid indirect testing where a piece of code may get full coverage without ever being intentionally tested.

2.3.3 Unit Test Scope

A unit test should be completely deterministic with respect to the code paths that it exercises. This means the test code must have control over all relevant inputs, i.e., inputs that influence the code paths. To illustrate, the current time is relevant when testing algorithms that perform date related functions, e.g., code that is conditioned on a certificate being expired, while it is not relevant if it is only used when printing log messages. Sources of non-determinism include time, random numbers, schedulers, and the network. A dependency on the time is typically mitigated by mocking the service that return the time. Random numbers can be handled the same way. A unit test should never sleep. Avoiding schedulers means a unit test should not have multiple processes and should not have multiple threads unless they cannot impact the code paths being tested. The network can be avoided by defining a suitable abstraction and mocking.

Code that relies on event dispatching may use a mock dispatcher to control the sequence of events. One design that makes it possible to unit test in this way is to organize a module as a set of atomic event handlers around a plain old data structure core. The core should be easy to test. Event handlers are called for timers, I/O readiness, and method calls into the module. Event handlers update the core and can perform I/O and call into other modules. Inter-module calls are problematic in that they create the possibility for deadlock and other hazards. In the simplest designs, each module has a single lock that is acquired at the beginning of each event handler. The non-deterministic part of the module can be tested by isolating its dependencies on the operating system and other modules; typically by providing mock objects.

To illustrate the other side of determinism, consider other kinds of tests. Integration tests often use operating system services, e.g., threads and networking, to test partial or whole system functionality. A stress test executes the same code over and over hoping that non-determism results in a different outcome. Performance tests may or may not admit non-determinism and focuses on aggregate behavior as opposed to code-level correctness. Unit tests should focus on code-level correctness.

2.3.4 Isolating Dependencies

More often than not, the code under test will have dependencies on other objects. For each dependency, the test can either pass in a real object or a stand-in. Test stand-ins have a variety of names including mocks, spies, dummies, etc. depending on their function. Some take the position that everything should be mocked. The author takes the position that real objects should be preferred for the following reasons:

- Less code to maintain
- The design of the real objects improves to accommodate testing
- Tests break in a more meaningful way when dependencies change, i.e., over time, a test stand-in may no longer behave in a realistic way

However, there are cases when a test stand-in is justified:

- It is difficult to configure the real object
- The real object lacks the necessary API for testing and adding it cannot be justified

The use of a mock assumes that an interface exists for the stand-in.

2.3.5 Writing a New Unit Test

1. Add the test to the appropriate file under tests/unit-tests.
2. Name the test after the code it is meant to cover. For example, the `tests/unit-tests/dds/DCPS/security/AccessControlBuiltInImpl.cpp` unit test covers the `dds/DCPS/security/AccessControlBuiltInImpl.(h|cpp)` files.
3. Update the tests/unit-tests/UnitTests.mpc file if necessary.

2.3.6 Using GTest

The main unit test driver is based on GTest. GTest provides you with many helpful tools to simplify the writing of unit tests. To use GTest in a test, add `#include <gtest/gtest.h>` to the unit test source file. A basic unit test has the following form

```
TEST(TestModule, TestSubmodule)
{
}
```

All tests in a unit test source file must have the same TestModule which is name of the unit under test with underscores, e.g., `dds_DCPS_security_AccessControlBuiltInImpl`. This naming convention is required for intentional unit test coverage. The TestSubmodule can be any identifier, however, it should typical describe the class, function, or scenario being tested.

Each test contains evaluators. The most common evaluators are `EXPECT_EQ`, `EXPECT_TRUE`, `EXPECT_FALSE`.

```
EXPECT_EQ(X, 2)
EXPECT_EQ(Y, 3)
```

This will mark the test as a failure if either `X` does not equal 2, or `Y` does not equal 3.

`EXPECT_TRUE` and `EXPECT_FALSE` are equivalence checks to a boolean value. In the following examples we pass `X` to a function `is_even` that returns true if the passed value is an even number and returns false otherwise.

```
EXPECT_TRUE(is_even(X));
```

This will mark the test as a failure if `is_even(X)` returns false.

```
EXPECT_FALSE(is_even(X));
```

This will mark the test as a failure if `is_even(X)` returns true.

There are more EXPECT_* and ASSERT_*, but these are the most common ones. The difference between EXPECT and ASSERT is that an ASSERT will cease the test upon failure, whereas EXPECTS continue to run. For example if you have multiple `EXPECT_EQ`, they will all always run.

For more information, visit the google test documentation: https://github.com/google/googletest/blob/master/docs/primer.md.

2.3.7 Code Coverage

To enable code coverage, one needs to disable the *dds_non_coverage* feature, e.g., `./configure ... --features=dds_non_coverage=0`.

The script `$DDS_ROOT/tools/scripts/unit_test_coverage.sh` will execute unit tests and generate an intentional unit test coverage report. It can be called with no arguments to generate a report for all of the units or it can be called with a list of units to test. For example, `$DDS_ROOT/tools/scripts/unit_test_coverage.sh dds/DCPS/Serializer`.

2.3.8 Final Word

Ignore anything in this document that prevents you from writing unit tests.

2.4 GitHub Actions Summary and FAQ

2.4.1 Overview

GitHub Actions is the continuous integration solution currently being used to evaluate the readiness of pull requests. It builds OpenDDS and runs the test suite across a wide variety of operation systems and build configurations.

2.4.2 Legend for GitHub Actions Build Names

Operating System

- u18/u20 - Ubuntu 18.04/Ubuntu 20.04
- w16/w19 - Windows Server 2016 (Visual Studio 2017)/Windows Server 2019 (Visual Studio 2019)
- m10 - MacOS 10.15

See also:

GitHub Virtual Environments

Build Configuration

- x86 - Windows 32 bit. If not specified, x64 is implied.
- re - Release build. If not specified, Debug is implied.
- clang5/clang10/gcc6/gcc8/gcc10 - compiler used to build OpenDDS. If not specified, the default system compiler is used.

Build Type

- stat - Static build
- bsafe/esafe - Base Safety/Extended Safety build
- sec - Security build
- asan - Address Sanitizer build

Build Options

- o1 - enables `--optimize`
- d0 - enables `--no-debug`
- i0 - enables `--no-inline`
- p1 - enables `--ipv6`
- w1 - enables wide characters
- v1 - enables versioned namespace
- cpp03 - `--std=c++03`
- j/j8/j12 - Default System Java/Java8/Java12
- ace7 - uses ace7tao3 rather than ace6tao2
- xer0 - disables xerces
- qt - enables `--qt`
- ws - enables `--wireshark`
- js0 - enables `--no-rapidjson`

Feature Mask

This is a mask in an attempt to keep names shorter.

- FM-08
  - `--no-built-in-topics`
  - `--no-content-subscription`
  - `--no-ownership-profile`
  - `--no-object-model-profile`
  - `--no-persistence-profile`
- FM-1f

    - `--no-built-in-topics`
- FM-2c
    - `--no-content-subscription`
    - `--no-object-model-profile`
    - `--no-persistence-profile`
- FM-2f
    - `--no-content-subscription`
- FM-37
    - `--no-content-filtered-topics`

2.4.3 build_and_test.yml Workflow

Our main workflow which dictates our GitHub Actions run is .github/workflows/build_and_test.yml. It defines jobs, which are the tasks that are run by the CI.

Triggering the Build And Test Workflow

There are a couple ways in which a run of build and test workflow can be started.

Any pull request targeting master will automatically run the OpenDDS workflows. This form of workflow run will simulate a merge between the branch and master.

Push events on branches prefixed `gh_wf_` will trigger workflow runs on the fork in which the branch resides. These fork runs of GitHub Actions can be viewed in the "Actions" tab. Runs of the workflow on forks will not simulate a merge between the branch and master.

Job Types

There are a number of job types that are contained in the file build_and_test.yml. Where possible, a configuration will contain 3 jobs. The first job that is run is *ACE_TAO_*. This will create an artifact which is used later by the OpenDDS build. The second job is *build_*, which uses the previous *ACE_TAO_* job to configure and build OpenDDS. This job will then export an artifact to be used in the third step. The third step is the *test_* job, which runs the appropriate tests for the associated OpenDDS configuration.

Certain builds do not follow this 3 step model. Safety Profile builds are done in one step due to cross-compile issues. Static and Release builds have a large footprint and therefore cannot fit the entire test suite onto a Github Actions runner. As a result, they only build and run a subset of the tests in their final jobs, but then have multiple final jobs to increase test coverage. These jobs are prefixed by:

- *compiler_* which runs the tests/DCPS/Compiler tests.
- *unit_* which runs the unit tests located in tests/unit-tests.
- *messenger_* which runs the tests in tests/DCPS/Messenger and tests/DCPS/C++11/Messenger.

To shorten the runtime of the continuous integration, some other builds will not run the test suite.

All builds with safety profile disabled and ownership profile enabled, will run the tests/cmake tests. Test runs which only contain CMake tests are prefixed by `cmake_`.

.lst Files

.lst files contain a list of tests with configuration options that will turn tests on or off. The *test_* jobs pass in tests/dcps_tests.lst. Static and Release builds instead use tests/static_ci_tests.lst. This seperation of .lst files is due to how excluding all but a few tests in the dcps_tests.lst would require adding a new config option to every test we didn't want to run. There is a seperate security test list, tests/security/security_tests.lst, which governs the security tests which are run when `--security` is passed to `auto_run_tests.pl`. The last list file used by `build_and_test.yml` is tools/modeling/tests/modeling_tests.lst, which is included by passing `--modeling` to `auto_run_tests.pl`.

To disable a test in GitHub Actions, `!GH_ACTIONS` must be added next to the test in the .lst file. These tests will not run when `-Config GH_ACTIONS` is passed alongside the lst file. There are similar test blockers which only block for specific github actions configurations from running marked tests:

- `!GH_ACTIONS_OPENDDS_SAFETY_PROFILE` blocks Safety Profile builds
- `!GH_ACTIONS_M10` blocks the MacOS10 runners
- `!GH_ACTIONS_ASAN` blocks the Address Sanitizer builds
- `!GH_ACTIONS_W16` blocks the Windows2016 runner

These blocks are necessary because certain tests cannot properly run on GitHub Actions due to how the runners are configured.

See also:

Running Tests For how `auto_run_tests.pl` works in general.

Blocked Tests

Certain tests are blocked from GitHub actions because their failures are either unfixable, or are not represented on the scoreboard. If this is the case, we have to assume that the failure is due to some sort of limitation caused by the GitHub Actions runners.

Only Failing on CI

- tests/DCPS/SharedTransport/run_test.pl multicast
  - Multicast times out waiting for remote peer. Fails on `test_u20_p1_j8_FM-1f` and `test_u20_p1_sec`.
- tests/DCPS/StringKey/run_test.pl
  - A timeout occurs during the writer writing. Fails on `u18_bsafe_js0_FM-1f`.
- tests/DCPS/Thrasher/run_test.pl high/aggressive/medium XXXX XXXX
  - The more intense thrasher tests cause consistent failures due to the increased load from ASAN. GitHub Actions fails these tests very consistently compared to the scoreboard which is more intermittent. Fails on `test_u20_p1_asan_sec`.
- tests/stress-tests/dds/DCPS/run_test.pl
  - This test fails due to only getting `17 of the expected >=19 total_count`. Fails on `test_m10_i0_j_FM-1f` and `test_m10_o1d0_sec`.
- tests/DCPS/StaticDiscoveryReconnect/run_test.pl
  - This test fails due to `<StaticDiscoveryTest> failed: No such file or directory`. Fails on `test_m10_i0_j_FM-1f` and `test_m10_o1d0_sec`.

Failing Both CI and scoreboard

These tests fail on the CI as well as the scoreboard, but will remain blocked on the CI until fixed. Each test has a list of the builds it was failing on before being blocked.

- tests/DCPS/BuiltInTopicTest/run_test.pl
  - u18_esafe_js0
- tests/DCPS/CompatibilityTest/run_test.pl rtps_disc
  - test_m10_o1d0_sec
- tests/DCPS/Deadline/run_test.pl rtps_disc
  - test_u20_p1_asan_sec
  - test_m10_o1d0_sec
- tests/DCPS/Federation/run_test.pl
  - test_u18_w1_sec
  - test_u18_j_cft0_FM-37
  - test_u18_w1_j_FM-2f
  - test_u20_ace7_j_qt_ws_sec
  - test_u20_p1_asan_sec
  - test_u20_p1_asan_sec
- tests/DCPS/Instances/run_test.pl [Multiple Configurations]
  - u18_bsafe_js0_FM-1f
  - u18_esafe_js0
- tests/DCPS/MultiDPTest/run_test.pl
  - u18_bsafe_js0_FM-1f
  - u18_esafe_js0
- tests/DCPS/NotifyTest/run_test.pl
- tests/DCPS/Reconnect/run_test.pl restart_pub
  - test_w16_x86_i0_sec
- tests/DCPS/Reconnect/run_test.pl restart_sub
  - test_w16_x86_i0_sec
- tests/DCPS/ReliableBestEffortReaders/run_test.pl
  - test_u18_w1_j_FM-2f
  - test_u18_j_cft0_FM-37
  - test_u20_p1_j8_FM-1f
  - test_m10_o1d0_sec
- tests/DCPS/TimeBasedFilter/run_test.pl -reliable
  - u18_bsafe_js0_FM-1f
  - u18_esafe_js0

Test Results

The tests are run using autobuild which creates a number of output files that are turned into a GitHub artifact. This artifact is processed by the "Check Test Results" workflow which modifies the files with detailed summaries of the test runs. After all of the Check Test Results jobs are complete, the test results will be posted in either the build_and_test or lint workflows. It is random which one of the workflows the results will appear in, so be sure to check both. This is due to a known problem with the GitHub API.

Artifacts

Artifacts from the continuous integration run can be downloaded by clicking details on one of the Build & Test runs. Once all jobs are completed, a dropdown will appear on the bar next to "Re-run jobs", called "Artifacts" which lists each artifact that can be downloaded.

Alternatively, clicking the "Summary" button at the top of the list of jobs will list all the available artifacts at the bottom of the page.

Using Artifacts to Replicate Builds

You can download the `ACE_TAO_` and `build_` artifacts then use them for a local build, so long as your operating system is the same as the one on the runner.

1. `git clone` the ACE_TAO branch which is targeted by the build. This is usually going to be `ace6tao2`.
2. `git clone --recursive` the OpenDDS branch on which the CI was run.
3. Merge OpenDDS master into your cloned branch.
4. run `tar xvfJ` from inside the cloned ACE_TAO, targeting the `ACE_TAO_*.tar.xz` file.
5. run `tar xvfJ` from inside the cloned OpenDDS, targeting the `build_*.tar.xz` file.
6. Adjust the setenv.sh located inside OpenDDS to match the new locations for your ACE_TAO, and OpenDDS. The word "runner" should not appear within the setenv.sh once you are finished.

You should now have a working duplicate of the build that was run on GitHub Actions. This can be used for debugging as a way to quickly set up a problematic build.

Using Artifacts to View More Test Information

Tests failures which are recorded on github only contain a brief capture of output surrounding a failure. This is useful for some tests, but it can often be helpful to view more of a test run. This can be done by downloading the artifact for a test step you are viewing. This test step artifact contains a number of files including `output.log_Full.html`. This is the full log of all output from all test runs done for the corresponding job. It should be opened in either a text editor or Firefox, as Chrome will have issues due to the length of the file.

Caching

The OpenDDS workflows create .tar.xz archives of certain build artifacts which can then be up uploaded and shared between jobs (and the user) as part of GitHub Actions' "artifact" API. A cache key comparison made using the relevant git commit SHA will determine whether to rebuild the artifact, or to use the cached artifact.

2.5 Running Tests

2.5.1 Main Test Suite

Building

Tests are not built by default, `--tests` must be passed to the `configure` script. This will build all the tests. There are a few ways to only have specific tests built:

- If using Make, specify the targets instead of leaving it default to the `all` target.
- Run MPC on the test directory and build separately. Make sure to also build the test's dependencies.
- Create a custom workspace with the tests and pass it to the `configure` script using the `--workspace` option. Also make sure to include the test's dependencies.

Running

Note: Make sure *ACE_ROOT* and *DDS_ROOT* are set, which can be done by running `source setenv.sh` on Linux and macOS or `call setenv.cmd` on Windows.

OpenDDS' main suite of tests is ran by the tests/auto_run_tests.pl Perl script that reads lists of tests from files and selectively runs based on how the script has been configured.

For Unixes (Linux, macOS, BSDs, etc)

Run this in *DDS_ROOT*:

```
./bin/auto_run_tests.pl
```

For Windows

Run this in *DDS_ROOT*:

```
bin\auto_run_tests.pl
```

If OpenDDS was built in Release mode add `-ExeSubDir Release`. If it was built as static libraries add `-ExeSubDir Static_Debug` or `-ExeSubDir Static_Release`.

Manual Configuration

Manual configuration is done by passing `-Config`, `-Exclude`, and test list files arguments to the script.

To manually configure what tests to run:

- See the `--list-all-configs` or `--show-all-configs` options to see the existing configurations used by all test list files.
- See the `--list-configs` or `--show-configs` options to see the existing configurations used by specific test list files.
- See the test list files for the tests themselves:
  - tests/dcps_tests.lst
    - This is included by default. Use `--no-dcps` to exclude this list.
  - tests/security/security_tests.lst
    - Use `--security` to include this list.
  - java/tests/dcps_java_tests.lst
    - Use `--java` to include this list.
  - tools/modeling/tests/modeling_tests.lst
    - Use `--modeling` to include this list.
- In a test list file each of the-space delimited words after the colon determines when the test is ran.
- Passing `-Config RTPS` will run tests that have `RTPS` and leave out tests with `!RTPS`.
- Passing `-Exclude RTPS` will exclude all tests that have `RTPS` in the entry. This option matches using RegEx, so a test with `SUPER_DUPER_RTPS` will also be excluded. It also ignores inverse entries, so it will not exclude a test with `!SUPER_DUPER_RTPS`.
- Assuming they were built, CMake tests are ran if `--cmake` is passed. This uses CTest, which is a system that is separate from the one previously described.
- See `--help` for all the available options.

2.6 Bench 2 Performance & Scalability Test Framework

2.6.1 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.

2.6.2 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.

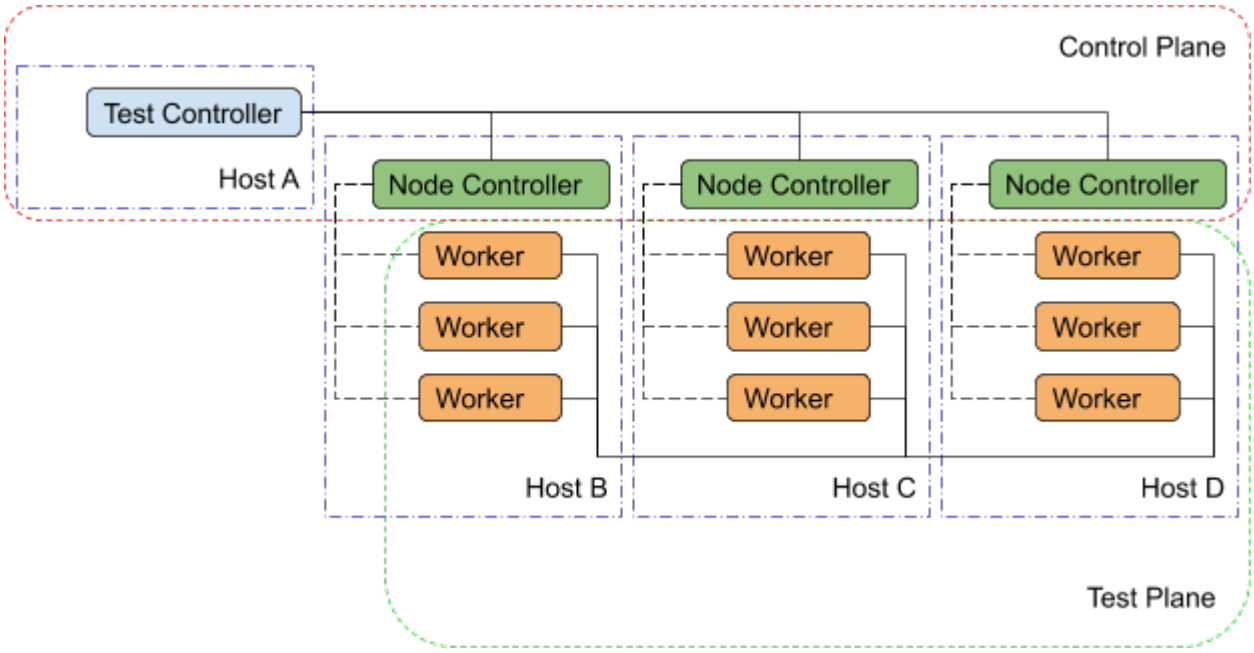

Fig. 2.1: Bench 2 Overview

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).

2.6.3 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`

2.6.4 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 performance-tests/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 performance-tests/bench/run_test.pl to execute the full scenario without first setting the environment as listed above. This is because 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. It 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_controller`s (across one or more hosts) and then execute the test_controller with the desired scenario configuration.

2.6.5 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 performance-tests/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
    }
  ],
```

(continues on next page)

(continued from previous page)

```
  "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:
```

(continues on next page)

(continued from previous page)

```
    DoubleSeq double_seq_prop;
  case PVK_ULL:
    unsigned long long ull_prop;
};

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

(continues on next page)

(continued from previous page)

```
// 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;

// 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 {
```

(continues on next page)

(continued from previous page)

```
  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;
  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;
```

(continues on next page)

(continued from previous page)

```
  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 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
            }
          ]
```

(continues on next page)

(continued from previous page)

```
        }
      ]
    }
  ]
},
"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",
```

(continues on next page)

(continued from previous page)

```
  "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 thought the list sequentially unless `"random_order"` flag (below) is specified

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

  ]
}
```

2.6.6 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

A

C

D

E

N

O

S

T

W