Date: July 2009

Proposal: MPI for CCA Components

Version (1.0)

CCA Document Number: P/2009-07-23-1

Standard document URL: To be determined

Necessary external references:

MPI-2: http://www.mpi-forum.org/docs/docs.html

Babel: https://computation.llnl.gov/casc/components/babel.html

cca.sidl: http://cca-forum.svn.sourceforge.net/viewvc/cca-forum/cca-spec/trunk/cca-spec-babel/cca.sidl?view=markup

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details a proposed Common Component Architecture Forum specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

PATENTS

No patented or patent-pending technology is discussed in this report.

GENERAL USE RESTRICTIONS

The specifications discussed in this document are provided to the public domain.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE CCA FORUM AND ANY COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE CCA FORUM OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

The United States Government retains all its usual and customary rights to this work as the work was entirely funded by the US Department of Energy.

TRADEMARKS

CCA Forum has not yet retained any trademarks. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners. “Common Component Architecture” and “CCA Forum” probably ought to be trademarked.

COMPLIANCE

The copyright holders listed above acknowledge that the CCA Forum (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by CCA Forum., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

CCA's Issue Reporting Procedure

All CCA specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page https://www.cca-forum.org/bugs/cca-spec-bugs.

Questions of a more general technical nature may be submitted via email to help@cca-forum.org.

Table of Contents

Preface

Common Component Architecture Forum

Founded in 1998, the CCA Forum (CCAF) is an open membership, not-for-profit scientific computing standards working group that produces and maintains scientific computing specifications for interoperable, portable, and reusable high-performance applications in distributed and parallel environments. Membership includes application vendors, end users, government agencies, and academia.

CCAF member organizations write, adopt, and maintain its specifications following a mature, open process. CCAF's specifications define the Common Component Architecture (CCA), providing a life-cycle approach to scientific software development that covers multiple operating systems, programming languages, middle-ware and networking infrastructures, and software development environments.

More information on the CCAF is available at http://www.cca-forum.org and http://www.tascs-scidac.org.

CCAF Specifications

As noted, CCAF specifications address middleware and scientific computing frameworks. The entire specification is available in the file cca.sidl from the CCA source repository website at:

http://sourceforge.net/projects/cca-forum

Specifications are written in the Scientific Interface Definition Language (SIDL) language as defined and supported by the Babel development team at Lawrence Livermore National Laboratory (LLNL). More information on SIDL is available from the Babel website at:

https://computation.llnl.gov/casc/components/babel.html

Supporting documentation for elements of the specifications is provided in the CCAF development records. The record of standards development documents and votes is available to CCAF members from the Quorum website at:

https://www.cca-forum.org/quorum

Products implementing CCAF specifications are available from individual suppliers.

Platform Specific Interface Specifications

All of CCAF's formal specifications are intended to be platform neutral. At present, however, the Babel (and hence CCA) software is not known to be available as a supported product except on UNIX-based platforms.

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: Scientific Interface Definition Language (SIDL) and syntax elements.

`Courier - 8 pt. :` Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

NOTE: Terms that appear in italics are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

1 Scope

This specification provides a CCA port definition giving components the ability to obtain an appropriately scoped MPI communicator from a framework service or another component.

2 Conformance

Support for the herein specified ports is optional for CCA framework implementations. When a framework or other implementation provides any port or other object with a SIDL type name rooted in the gov.cca package, that implementation shall follow the specifications in cca.sidl. Implementors shall not define type names starting with "gov.cca." unless the type names appear in the approved cca.sidl.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

None.

4 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply. Terms clearly taken from the MPI specification (typically prefixed with MPI) are not redefined here.

Collective

Precisely as defined in the MPI-2 Specification Semantic Terms. Roughly, an action taken identically on all members of a process group.

Communicator

An instance of MPI_Comm, represented as a SIDL long (8 byte integer) in all language bindings. Component implementations and clients may need to cast this value to or from the appropriate integer width and convert to or from the language-specific MPI handle type.

SCMD

Single component, multiple data programming model, derived from the more general single program, multiple data programming model. The most common case of SCMD is for a parallel application to have a single instance of a CCA framework in each of the processes in an MPI job and for identical framing actions (component creation, component connection) to be performed on all processes.

MCMD

Multiple component, multiple data programming model, derived from the more general multiple program, multiple data programming model. The most common case of MCMD is for a parallel application to have a multiple MPI process groups and for each group to be constructed and behave (except for periodic inter-group data exchanges) as a SCMD program.

Service

A uses port available for use via getPort automatically without requiring user-directed port connections. A service port is defined in the framework by inserting the server in the framework-provided ServiceRegistry port. Connections are made for services when matched based on port SIDL type. The critical distinction between a service and a regular uses port is that services are available during the gov.cca.Component:setServices call, but regular uses ports require additional connection information which is not available at that time.

5 Symbols

None.

6 Additional Information

6.1 Changes to Adopted CCA Specifications

None.

6.2 Acknowledgements

The following companies submitted this specification:

- Sandia National Laboratories

The following framework developers have performed validation tests of this specification:

Benjamin Allan

Kostadin Damevski

7 Specification *MPIService*

This section presents the specification for gov.cca.ports.MPIService. It begins with an overview of the intended uses.

7.1 Overview *Support for MPI-based components*

A recommended "best practice" for MPI-based applications is that a component (or any library intended to be reusable) should not assume it knows the parallel context in which it is being used. More specifically, it should use an MPI communicator obtained from some external (to the component) source rather than using MPI_COMM_WORLD. Additionally, in general, each component should utilize a different MPI communicator in order to preclude conflicts in message delivery.

There are two basic ways for a component to obtain a communicator from an external source. In the "push" approach, a caller passes the communicator to the component as an argument in a method call. A "pull" approach requires the component to query a port connected to an external source to get the communicator.

The MPIService port defines a standard representation for the pull method of obtaining a communicator. The MPIService port also allows components to obtain rank and size information without actually producing a new communicator, which can be expensive (depending on the underlying MPI implementation). This supports components that do not use MPI directly, but use size and rank information for other purposes, such as data decomposition, reporting, debugging, and file naming.

For a component following the "push" model, the MPIService port could be used by a second managing component to obtain a communicator which is then passed as an argument to the push-oriented component.

Automating connections to an MPIService provider supports the common case where a framework is used to construct an application following the single-program multiple-data (SPMD) architecture. A framework's ServiceRegistry can be configured to provide these automatic connections. A component writer using such a framework can assume MPIService is available to the component during the setServices call of component construction, thus guaranteeing MPI reso urces are obtained before any client has an opportunity to call on the component.

While MPIService is most commonly thought of as a SPMD framework service, it may also be provided by standalone components. This allows, for example, applications requiring complex sets of communicators to use multiple MPIService components to provide them. In this case, of course, due care must be taken when assembling the application to insure that components are connected to the appropriate MPIService instances.

Several components all from the same vendor may be coded and optimized such that the vendor's components within a single process group can share a single communicator without message delivery conflicts. MPIService is not appropriate for use in this design. Creating an analogous service interface for sharing a single communicator instance at setServices time (call it MPIBorrow for discussion purposes) is possible under any predefined scheme assigning message tags to components. However, a general, vendor-neutral MPIBorrow definition of such a tag assignment scheme is beyond the scope of this specification.

7.1.1 Proposed SIDL interface gov.cca.ports.MPIService

```
 /** GENERAL NOTES on MPI handles and SIDL:
*
* As MPI Forum explicitly DOES NOT define an interlanguage
* form for object (comm, group, etc) handles, we use the
* FORTRAN form which, being an integer of some size <= long
* in sidl, we will express in SIDL as a long.
*
* Implementations in C/C++/Python will have to use appropriate
* conversion operators, MPI_Comm_c2f/MPI_Comm_f2c, when
* passing/receiving object handles in long form.
*
* The MPI standard does not include an automatic reference counting
* model for its handles.
*/

   interface MPIService extends gov.cca.Port {

     /** Get an mpi communicator with the same process group as the provider
      instance and with the intended process group of the caller.
      Result will not be mpi_comm_null. The communicator returned will be
      private to the recipient, which implies an mpicommdup by the provider.
      Call must be made collectively.
      @return The comm produced, in FORTRAN form. C/C++ callers use comm_f2c
            method defined by their mpi implementation, usually MPI_Comm_f2c,
            to convert result to MPI_Comm.
      @throw CCAException if the service cannot be implemented because MPI is
           not present.
      */
     long getComm() throws gov.cca.CCAException;

     /** Let go of the communicator previously fetched with getComm.
      * Call must be made collectively.
      *  @throw CCAException if an input error is detected.
      */
     void releaseComm(in long comm) throws gov.cca.CCAException;

     /** Get the typically needed basic parallelism information for a component that
      *  requires no MPI communication and thus does not need an independent communicator.
      * Rationale: on very large machines, the cost of a Comm_dup should be avoided where possible;
      * the other calls on a MPI Comm object may affect its state, and thus should not
      * be proxied here.
      *  @throw CCAException if an error is detected.
      */
     void getSizeRank(out long commSize, out long commRank) throws gov.cca.CCAException;

   } // end MPIService
```

Figure 1 - gov.cca.ports.MPIService SIDL definition.

7.1.2 Advice to implementors

Implementors are advised to consider reusing the component-based MPIService implementation referenced in 7.3.2 and Annex A. If creating a new MPIService implementation that is expected to run in the absence of an initialized MPI, the recommended value to return from getComm (along with the prescribed exception) is 0. For this same MPI-absent case, the appropriate outputs of getSizeRank for commSize and commRank are 1 and 0, respectively.

7.2 Programming models *SCMD and MCMD use of MPIService*

The MPIService port may be served by the framework itself, in which case all MPIService users will receive communicators which are MPI_CONGRUENT with the communicator of the parallel framework instance. In most cases a SCMD application will result.

MCMD applications may be framed from SCMD components assuming MPIService and the existing specifications in multiple ways. One is for the driving application to split MPI_COMM_WORLD into separate communicators and then initialize a separate parallel framework instance for each process group. This scenario is illustrated in Annex A.3. A more complex way is for the driver of a single parallel framework instance initialized with a duplicate of MPI_COMM_WORLD to replace (by use of the ServiceRegistry port) the framework provided implementation of MPIService with an implementation serving an alternative communicator scope.

7.3 Prototype feasibility tests

The proposed service has run successfully in multiple frameworks with multiple implementations, using a different sidl package name (dev.cca.ports.MPIService) to distinguish it from approved standards (gov.cca.*). If MPIService is adopted, all references in this document to dev.cca.ports.MPIService will be replaced with gov.cca.ports.MPIService.

7.3.1 Ccaffeine Implementation

Service interface ccaffeine.ports.MPIService has existed since 1999. It has been used in applications since that time and was ported to SIDL when SIDL support was added to Ccaffeine.

7.3.2 Component-based implementation

Service dev.cca.ports.MPIService and a supporting implementation component and test component are framework independent components (requires only that the framework supports ServiceRegistry and a platform supporting MPI) are available at:
http://cca-forum.svn.sourceforge.net/viewvc/cca-forum/cca-tutorial/trunk/pde-hands-on/src-mpi/

These components have been tested successfully in the Ccaffeine framework configured with or without its default ccaffeine.ports.MPIService. Examples from the repository code are provided in the informative annexes.

7.3.3 SCIJump implementation

Prototype service dev.cca.ports.MPIService has been ported to and tested in the SCIJump framework by reuse of the component-based implementation.

7.4 Evolution with MPI Specifications

This specification is for use with MPI-2 and any other version of MPI which is consistent with the Fortran integer & C void * definition of MPI object handles. Should an incompatible change occur in a newer MPI, a new CCAF specification (e.g. MPI3Service) will be in order.

Annex A: SIDL and C++ implementation examples

(Informative)

Normative annexes are integral parts of the standard. Their presence is optional. An annex's normative status (as opposed to informative) shall be made clear by the way in which it is referred to in the text and under the heading of the annex.

Informative annexes give additional information intended to assist the understanding or use of the standard and shall not contain provisions to which it is necessary to conform in order to be able to claim compliance with the standard. Their presence is optional. An annex's informative status (as opposed to normative) shall be made clear by the way in which it is referred to in the text and under the heading of the annex.

A.1 MPIService implemented as a component

The MPISetup port (Figure 3) demonstrates setup functions appropriate for a component implementing MPIService suitable for SCMD or MCMD application; it is not part of the proposed specification. The dev.cca.common.MPICommSource class (Figure 2) is a component implementing MPIService and MPISetup; it is not part of the specification.

```
class  dev.cca.common.MPICommSource
implements-all gov.cca.Component, gov.cca.ComponentRelease, dev.cca.ports.MPIService,
dev.cca.ports.MPISetup
{
// ports provided: MPIService, MPISetup
// ports used: gov.cca.ports.ServiceRegistry
// other types required: an implementation of CCAException, and all types referred to directly in
MPISetup and MPIService
}
```

Figure 2 - dev.cca.common.MPICommSource SIDL definition.

The MPISetup interface and the MPICommSource implementation do not support the assumption that the initialization of MPI itself should be hidden inside a CCA component. The definition of MPISetup is new, is not widely used, is likely to be used in the future only by a small number of framework developers, and is predicated on a MPI_Init being called outside the component environment. Therefore MPISetup is not proposed as part of the standard at this time.

```
interface MPISetup extends gov.cca.Port
{
    /** Check instance status. Only one init* call per instance is allowed.
        @return true if initAsService or initAsComponent or initAsInstance is already done.
    */
    bool isInitialized();

    /**
    *
    * This method is for treating an instance from an external driver
    * to set up a general service instance global to a frame. This is useful in at least the static linking case.
    *
     * Create and add to the framework MPIService
     * support. This will appear in the frame as an
     * MPICommSource component instance without necessarily existing
     * in the BuilderService accessible class list.
     * MPI_Init must have been called before this is called.
     * This entry point should work for any CCA framework bootstrapping
     * in MPI_COMM_WORLD or an otherwise scoped communicator via the standard
     * ServiceRegistry interface. This will not automatically
     * cause the component class providing this interface to appear in the
     * list of classes the user may instantiate.
     * In the MPI sense, this call must be collective on the scope of dupComm.
     *
     * @param dupComm  the (valid) communicator (in fortran form) to duplicate
     *   for those using MPIService.
     * @param af The frame into which the server will add itself.
     * In principle, the caller should be able to forget about the class object they are holding to make this call.
     */
    void initAsInstance(in long dupComm, inout gov.cca.AbstractFramework af) throws gov.cca.CCAException;

    /** Set the communicators on an uninitialized MPI support component
     * instance created like any other and register the component through
     * the ServiceRegistry.
     *
     * In the MPI sense, this call must be collective on the scope of dupComm.
     *
     * @param dupComm  the (valid) communicator (in fortran form) to duplicate
     *   for those using MPIService.
     */
    void initAsService(in long dupComm);

    /** Set the communicators on an uninitialized mpi support component
     * instance created like any other.
     * This does NOT cause the component being initialized to register itself
     * as a service for all comers.
     * This method is for treating an instance from inside a frame or
     * subframe as a peer component that may serve only certain other components in the frame, e.g after a comm split.
     *
     * In the MPI sense, this call must be collective on the scope of
     * dupComm.
     *
     * @param dupComm  the (valid) communicator (in fortran form) to duplicate
     *   for those using MPIService.
     */
    void initAsComponent(in long dupComm);

    /**
     * Shutdown the previous mpi-related services.
     * @param reclaim if reclaim true, try to release communicator
     * resources allocated in MPIService support.
     * Otherwise, leak them. Not reclaiming is bad practice, but may be required to work around improperly code MPI-using
     * components.
     */
    void finalize(in bool reclaim) throws gov.cca.CCAException;
}
```

Figure 3 - dev.cca.ports.MPISetup SIDL definition.

A.2 C++ example of using MPIService methods in a SCMD component

```
/* from class header */
    // DO-NOT-DELETE splicer.begin(demo.TestMPI._implementation)
    gov::cca::Services d_services;
    gov::cca::Port msp;
    MPI_Comm cComm;
    void whine(std::string s, const char *fname, int lineno, const char *function )
    {
     sidl::SIDLException ex = ::sidl::SIDLException::_create();
     ex.setNote(s); ex.add(fname, lineno, function);
     throw ex;
    }
#define WHINE(s,f) whine(s,__FILE__,__LINE__,f)
    // DO-NOT-DELETE splicer.end(demo.TestMPI._implementation)

/* from class code */
void demo::TestMPI_impl::setServices( ::gov::cca::Services services )
{
  //  DO-NOT-DELETE splicer.begin(demo.TestMPI.setServices)
  d_services = services;
  gov::cca::TypeMap tm;
  d_services.registerUsesPort("MPIService","gov.cca.ports.MPIService",tm);
  try {
    msp = services.getPort("MPIService");
    gov::cca::ports::MPIService ms = ::babel_cast< gov::cca::ports::MPIService >(msp);

    int64_t size = -1, rank = -1;
    ms.getSizeRank( size, rank );

    int64_t myComm = ms.getComm();
    if (myComm == 0) {
        WHINE("found 0 for MPIService (shocking!)",  demo.TestMPI.setServices  );
    }
    MPI_Fint fmComm = (MPI_Fint)myComm;
    cComm = MPI_Comm_f2c(fmComm);
    if (cComm == MPI_COMM_NULL) {
        WHINE("found MPI_COMM_NULL from MPIService",   demo.TestMPI.setServices  );
    }
   int m_err  = MPI_Barrier(cComm); // example mpi call.
    // ought to check m_err == MPI_SUCCESS; cannot be sure errors will abort.
  } catch ( sidl::SIDLException ex ) {
     ex.add(__FILE__, __LINE__,   demo.TestMPI.setServices  );
  }
  //  DO-NOT-DELETE splicer.end(demo.TestMPI.setServices)
}

void demo::TestMPI_impl::releaseServices( ::gov::cca::Services services)
{
  //  DO-NOT-DELETE splicer.begin(demo.TestMPI.releaseServices)
  if (msp._not_nil()) {
    MPI_Fint fmcomm = MPI_Comm_c2f(cComm);
    int64_t myComm = fmComm;
    try {
      gov::cca::ports::MPIService ms = ::babel_cast< gov::cca::ports::MPIService >(msp);
      ms.releaseComm(myComm);
    } catch ( sidl::SIDLException ex) {
      std::cerr <<   Ignoring:   << ex.getNote() << std::endl;
    }
    d_services.releasePort(  MPIService  );
  }
  d_services.unregisterUsesPort("MPIService");
  // DO-NOT-DELETE splicer.end(demo.TestMPI.releaseServices)
}
```

Figure 4 - Simple use of MPIService: C++ calling portably on C binding of MPI

A.3 C++ example of using separate framework instances for MCMD

This example uses illustrates the simplest MCMD usage of MPIService with Ccaffeine. There are two MPIServices involved for each process: one in the top (MCMD) framework instance spanning all processes and one in the SCMD framework instance. Complete supporting code for this example is available in the CCA tutorial repository in the src-mpi module. The driver code is presented in Figures 5, 6, and 7, and graphically illustrated in Figure 8.

```
#include "dc/babel.12/babel-cca/AllBabelCCA.hxx"
#include <mpi.h>
#include "mpi/dev_cca_ports_MPISetup.hxx"
#include "mpitest/demo_SCMDComponentSetup.hxx"

/* One of the goals of CCA standards is to make the simple things easy and the
   hard things possible.  MPMD coding is relatively hard.
 */
int main(int argc, char **argv)
{
  MPI_Init(NULL,NULL);
  {
  ccaffeine::AbstractFramework caf = ccaffeine::AbstractFramework::_create();
  gov::cca::TypeMap nulltm;
  std::string args = "--path /home/baallan/cca/sf/pde-hands-
on/obj/mpiinstall/share/cca:/home/baallan/cca/sf/pde-hands-on/obj/mpiinstall/lib";
  caf.initialize(args,0,false); // set up ccaffeine without default internal mpiservice support.
  gov::cca::AbstractFramework top = caf;  // ccaffeine dependency ends here.

  // Get a Services handle for main() acting as a component 'main' in frame 'top'.
  // Its phantom sidl class name is mcmd.main.
  gov::cca::Services topServices = top.getServices("main","mcmd.main", nulltm);
  topServices.registerUsesPort("topBuilder","gov.cca.ports.BuilderService", nulltm);
  gov::cca::Port port = topServices.getPort("topBuilder");

  // get BuilderService and load an mpiservice implementation in the top frame
  gov::cca::ports::BuilderService topBuilderService = ::babel_cast< gov::cca::ports::BuilderService
>(port);
  gov::cca::ComponentID topframeComponentID = topBuilderService.getComponentID("main");
  gov::cca::ComponentID mpiTopComponentID = topBuilderService.createInstance("MPITop",
"dev.cca.common.MPICommSource", nulltm);

  // grab the MPISetup port and initialize it with the world communicator.
  topServices.registerUsesPort("MPISetup", "dev.cca.ports.MPISetup", nulltm);
  gov::cca::ConnectionID mpiSetupConnID = topBuilderService.connect(topframeComponentID,"MPISetup",
mpiTopComponentID, "MPISetup");
  port = topServices.getPort("MPISetup");
  dev::cca::ports::MPISetup topMPISetup = ::babel_cast< dev::cca::ports::MPISetup >(port);
  // note excruciatingly explicit casting
  MPI_Fint top_fortran_comm = MPI_Comm_c2f(MPI_COMM_WORLD);
  int64_t top_sidl_comm = (int64_t)top_fortran_comm;
  topMPISetup.initAsService( top_sidl_comm );
  // frame top now has a running MPIService that was loaded as a component.

  // Now in the top frame create some application specific data interchange component
  // It will use the MPIService in the top frame.
  // Eventually it will also appear as a component in each of the scmd subdomains.
  gov::cca::ComponentID topDataBridgeComponentID = topBuilderService.createInstance("DataBridge",
"demo.DataBridge", nulltm);

// continued in next figure.
```

Figure 5 – MCMD application part 1: top frame setup

The example can be generalized to any number of non-overlapping process groups by making the appropriate MPI split.

```
// continued from previous figure
  // for simple mcmd use, we construct scmd subdomains with non-overlapping communicators.
  gov::cca::AbstractFramework scmd = top.createEmptyFramework();

  // set the main program up as a component in the scmd frames.
  gov::cca::Services scmdServices = scmd.getServices("submain","scmd.main",nulltm);

  // split the mpi world naively into subdomains 'ODDS' and 'EVENS'.
  // This example can work with any number of subdomains (non-overlapping); it is not limited to 2
subdomains.
  int worldrank = -1;
  MPI_Comm_rank(MPI_COMM_WORLD, &worldrank);
  MPI_Comm scmd_comm;
  MPI_Comm_split(MPI_COMM_WORLD, worldrank % 2, worldrank, &scmd_comm);

  scmdServices.registerUsesPort("scmdBuilder","gov.cca.ports.BuilderService", nulltm);
  port = scmdServices.getPort("scmdBuilder");

  // get BuilderService and load an mpiservice implementation in the scmd frames
  gov::cca::ports::BuilderService scmdBuilderService = ::babel_cast< gov::cca::ports::BuilderService
>(port);
  gov::cca::ComponentID scmdframeComponentID = scmdBuilderService.getComponentID("submain");
  gov::cca::ComponentID mpiScmdComponentID = scmdBuilderService.createInstance("MPIScmd",
"dev.cca.common.MPICommSource", nulltm);

  // grab the scmd frame MPISetup port and initialize it with the scmd subdomain communicator.
  scmdServices.registerUsesPort("MPISetup", "dev.cca.ports.MPISetup", nulltm);
  gov::cca::ConnectionID scmdMPISetupConnID =
scmdBuilderService.connect(scmdframeComponentID,"MPISetup", mpiScmdComponentID, "MPISetup");
  port = scmdServices.getPort("MPISetup");
  dev::cca::ports::MPISetup scmdMPISetup = ::babel_cast< dev::cca::ports::MPISetup >(port);
  // note excruciatingly explicit casting again
  MPI_Fint scmd_fortran_comm = MPI_Comm_c2f(scmd_comm);
  int64_t scmd_sidl_comm = (int64_t)scmd_fortran_comm;
  scmdMPISetup.initAsService( scmd_sidl_comm );
  // frame scmd now has a running MPIService that was loaded as a component.


  // Now we make the bridge component created in the top frame appear in the scmd frame.
  // This is just like setting up main to act as a component in both the top and scmd frames.
  gov::cca::Services scmdDataBridgeServices = scmd.getServices("scmdDataBridge", "demo.DataBridge",
nulltm);

  // We tell the DataBridge about the second (sub)framework instance
  // that it is participating in by calling a method DataBridge defines on a port it
  // provides to the top frame.
  topServices.registerUsesPort("SCMDComponentSetup", "demo.SCMDComponentSetup", nulltm);
  gov::cca::ConnectionID scmdComponentSetupConnID =
topBuilderService.connect(topframeComponentID,"SCMDComponentSetup", topDataBridgeComponentID,
"SCMDComponentSetup");
  port = topServices.getPort("SCMDComponentSetup");
  demo::SCMDComponentSetup scmdComponentSetup = ::babel_cast< demo::SCMDComponentSetup >(port);
  scmdComponentSetup.setServicesInSubdomain(scmdDataBridgeServices);
  // The databridge now appears as a component in each of the scmd subdomains.

  /// now frame and run something in the scmd frames. Exactly what depends on your subdomain.
  if (worldrank % 2) {
    // subdomain ODDS. use scmdBuilderService to do whatever you want in this domain.
    // scmd components using mpiservice will automatically get the correct communicators.
  } else {
    // subdomain EVENS. use scmdBuilderService to do whatever you want in this domain.
  }

// continued in next figure.
```

Figure 6 – MCMD application part 2: SCMD subdomain frame setup

The corresponding finalization code for all the previous steps is needed to avoid errors on exiting

```
// continued from previous figure
  // compute done. now tear it all down in reverse.

  // shutdown databridge ports in scmd frame and disconnect from it in top frame.
  scmdComponentSetup.releaseServicesInSubdomain(scmdDataBridgeServices);
  topServices.releasePort("SCMDComponentSetup");
  topBuilderService.disconnect(scmdComponentSetupConnID,0);
  topServices.unregisterUsesPort("SCMDComponentSetup");

  // deregister databridge from scmd frame.
  scmd.releaseServices(scmdDataBridgeServices);
  // shut down mpiservice in scmd frame and destroy mpisetup server there
  scmdMPISetup.finalize(false);
  MPI_Comm_free(&scmd_comm);
  scmdServices.releasePort("MPISetup");
  scmdBuilderService.disconnect(scmdMPISetupConnID, 0 );
  scmdServices.unregisterUsesPort("MPISetup");
  scmdBuilderService.destroyInstance(mpiScmdComponentID,0);

  // disconnect from scmd builder and close scmd frame
  scmdServices.releasePort("scmdBuilder");
  scmdServices.unregisterUsesPort("scmdBuilder");
  scmd.releaseServices(scmdServices);
  scmd.shutdownFramework();


  // destroy databridge in top frame
  topBuilderService.destroyInstance(topDataBridgeComponentID, 0);
  // shut down mpiservice in top frame and destroy mpisetup server there
  topMPISetup.finalize( false );
  topServices.releasePort("MPISetup");
  topBuilderService.disconnect(mpiSetupConnID, 0);
  topServices.unregisterUsesPort("MPISetup");
  topBuilderService.destroyInstance(mpiTopComponentID, 0);

  // disconnect from top builder and close top frame
  topServices.releasePort("topBuilder");
  topServices.unregisterUsesPort("topBuilder");
  top.releaseServices(topServices);
  top.shutdownFramework();

  // The dtor on everything needs to be finished before MPI_Finalize,
  // so we close the scope. In other languages, we would need deleteref on
  // the mpisetup port instances we're holding. Else mpich will exit(1)
  // if shutdown deallocates lingering references after the return 0 here.
  }
  MPI_Finalize();

  return 0;
}
```

Figure 7 – MCMD application part 3: correct tear down of the framework instances and MPI.

Figure 8 provides a visual summary of the MCMD application structure using a slightly extended version of the Ccaffeine GUI notation. The extensions are: the normally invisible framework services are rendered as ports attached to the frame wall, component instances defined in the main via gov.cca.AbstractFramework.getServices instead of as normally through BuilderService are shown with dotted outlines, and instances which share state across frames are indicated with arrows and ovals.

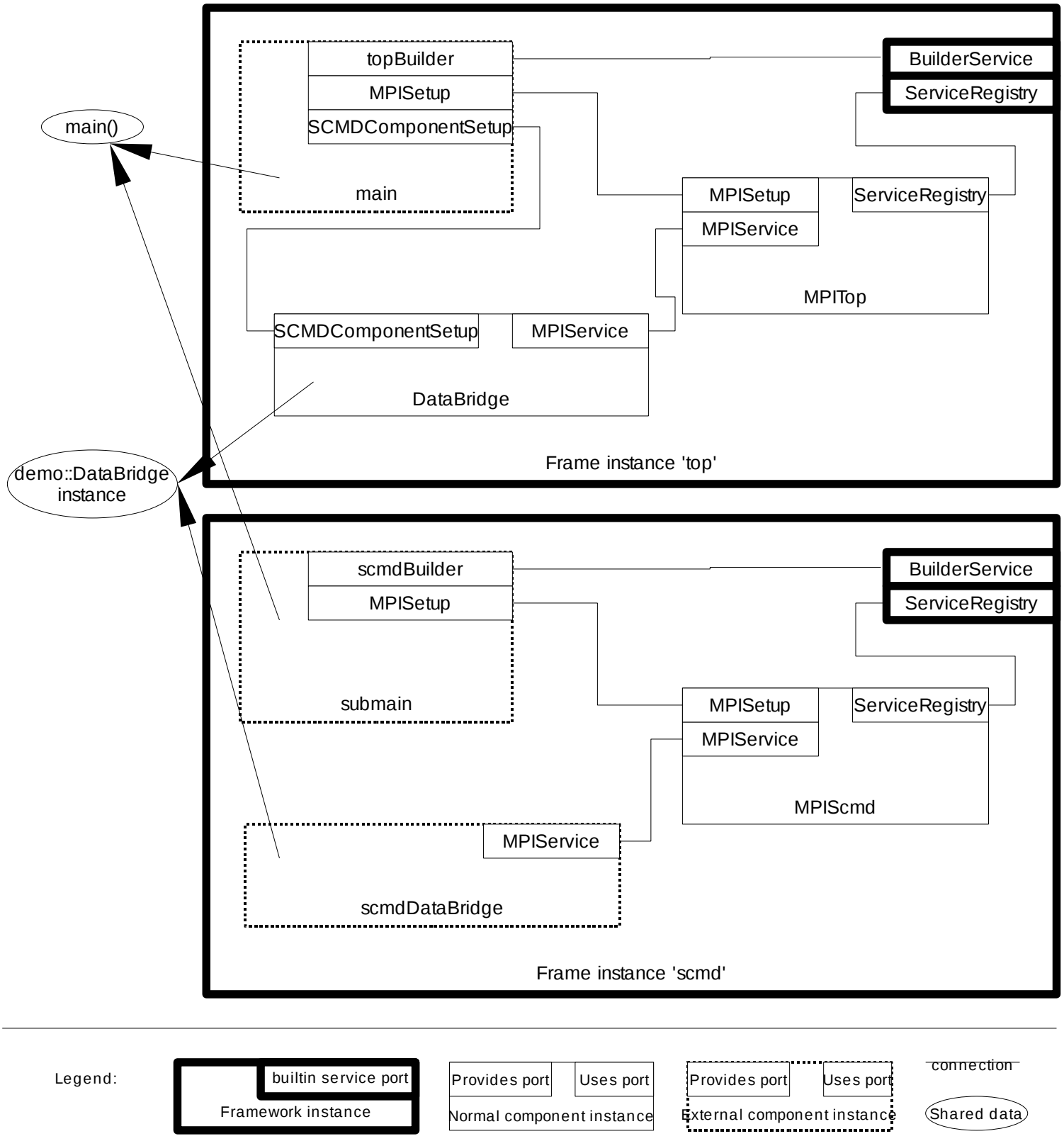

Figure 8 - CCA GUI notation equivalent to figures 5, 6, 7