Saydx specification

Release 0.1

B. Aradi, B. Hourahine

Jan 13, 2022

Contents

CHAPTER 1

Introduction

There is a clear need for a general and robust transport protocol to enable data exchange and communication between scientific software packages that need to interact. To suport this and demonstrate its use, a library implementation of the protocol should be developed and offered to the general scientific computing community. The proof of concept application for the library should be demonstrated for atomistic simulation packages, but the protocol and library need to be general enough to satisfy the needs of other kind of scientific software as well.

There are a number of methods for communication between codes. However are either special purpose implementations or do not abstract this task for the developers of the communicating codes.

1.1 Main goals

- Data exchange should be robust, guaranteeing reliable transmission.
- One-to-one and eventually many-to-one and many-to-many communication scenarios should be supported.
- Exchange of complex data (e.g. all the information needed to initialize and start a simulation) should be straightforward.
- The protocol should allow communication through different communcation channels.
- Cross language support for Fortran, C/C++ and Python family langauages with cross-platform numerical model support.

1.2 Communication layers

In order to ensure flexibility, the data exchange protocol needs (probably) three layers of implementation:

1. Transport layer: deals with the technicalities of the communication. It should allow multiple transport channels, e.g. file I/O, socket communication, loadable code modules, etc. It should be extensible for future channels and guarantee communication reliability.

2. Message layer: Provides a flexible message format, which can be transmitted through the low-level layer betwen the applications.
3. High-level communication layer: Domain specific protocol composed of messages, as customised for the scientific codes' using the library.

1.2.1 Transport layer

It would be good if many different transport channels can be supported and they are treated on the same footing. It would be desirable, if data could be exchanged via

- File I/O (text and binary)
- Socket communication
- MPI-messaging
- Binary API (enabling direct communication between C, Fortran, Python, etc. programs)

For the socket and MPI channels, we could probably directly use Steve Plimptons cslib library, which uses the quite robust ZeroMQ framework for the socket communication. It is already part of LAMMPS but could also be used as a stand-alone library without LAMMPS. Alternatively, one could write something similar.

1.2.2 Message layer

One should use messages that are flexible enough to carry complex information. For scientific applications the exchange of array data seems to be enough, provided several arrays can be sent as one message and different data types are supported within a message.

Example: Driver (a molecular-mechanics (MM)-program) sends data to a calculator (quantum-mechanical (QM)-program) to initialize it. This can be quite complex, as QM-programs usually require a lot of initialization parameters (Hamiltonian settings, basis set settings, various control settings, etc.). The message format needs to be flexible enough to allow for optional components, so that the driver has to specify only required settings and also optional ones which it wishes to override.

This could be easily realized by using a data tree as a message. A possible structure could be like a simplified XML DOM-tree with following specification:

- Each node of the tree is named (like in XML).
- Each node of the tree can either contain further nodes (a container node) or data (data nodes), but never both. Consequently, data nodes were the leaves of the tree and have only container nodes as parents.
- Each data node contains a single array of a given type and shape or a scalar as data. The data is in native binary format.
- Optional: the nodes should contain attributes to store additional information (e.g. the unit of the data in the node, etc.). To make things simple, the attributes should be text attributes, like in XML.

The sender would assemble a tree with the necessary information and transmit it via the transport layer. The receiver would then query the received tree, look for the presence / absence of given nodes and extract the necessary information from the tree.

I have already started a small C-library with this functionality, the saydx [sedks] library. Although not finished yet, it could be used as the message layer. It would provide the basic infrastructure for tree manipulation, as well as routines to read and write trees to file or to pass them from C to Fortran and vice-versa. Combined with cslib, it could cover the functionality of the first two layers.

Array types

The message layer should understand / support following array data types:

- real numbers (4, 8 and eventually 16 bytes)
- complex numbers (composed of two real numbers)
- integers (4 and 8 bytes)
- logicals (represented by integers)
- characters (1 byte)
- strings (of arbitrary length)
- bytes

When the data is represented in binary form, the native representation of the x86_64 architecture should be used. So, at least in its first version, the binary version of the protocol won't be architecture independent. We could allow for passing the arrays also in text form if this ever becomes an issue. (The saydx library can already store the tree in text form.) Implementing architecture independence on the binary level (as in HDF5) would be probably an overkill.

Array indexing

The arrays should be stored in the row-major format. The data should never be reordered in the message layer. In order to ensure, that normal indexing techniques (row-major in C and Python, column-major in Fortran) allow a continuous traversal in memory, the indexing tuple (but not the data!) should be reversed when the array shape is queried in a column-major language.

1.2.3 Protocol layer

In contrast to the other two layers, the protocol layer must be domain specific, as different scientific applications need different data to be communicated.

As a proof of concept, communication between atomistic simulation packages could be implemented. One could start from the i-Pi protocol, as several packages are using it already, base it on the new message format and extend it with additional components.

As an example, the transmitted data for passing the geometry between driver and client could look like the structure sketched below. The XML-notation is used to indicate nodes and the `@` symbols indicate (binary) scalars or arrays of a given type and shape in the leaves (e.g., `@s` is scalar string, `@r8(2,3)` is a rank two array of 64 bit reals with shape (2, 3), etc.):

```
<ipi-message>
  <command>
    @s
    POSDATA
  </command>
  <data>
    <atom_positions>
      @r8(2,3)
      0.0   0.0   0.0
      0.0   0.0   1.0
   </atom_positions>
   <lattice_vectors>
      @r8(3,3)
      10.0  0.0  0.0
```

(continues on next page)

(continued from previous page)

```
       0.0 10.0  0.0
       0.0  0.0 10.0
    </lattice_vectors>
 </data>
<ipi-message>
```

The receiver could then query the transmitted tree using following Fortran pseudo code:

```
call receive_tree(root_node)
if (root_node%get_name() /= "ipi-message") then
  call error("Invalid message protocol")
end if

call get_child_data(root_node, "command", commandstr)
if (.not. allocated(commandstr)) then
  call error("Could not find command node or it contains wrong data type")
end if

select case (commandstr)

case ("POSDATA")

  call get_child(root_node, "data", data_node)
  if (.not. data_node%is_associated()) then
    call error("Data node not found")
  end if

  call get_child_data(data_node, "atom_positions", atom_positions)
  if (.not. allocated(atom_positions)) then
    call error("Node 'atom_positions' not found or it contains wrong data type")
  end if
  if (all(shape(atom_positions) /= (3, natom)) then
    call error("Array in node 'atom_positions' has invalid shape")
  end if

  ! Only query tree for lattice vectors if the system is periodic
  if (periodic) then

    call get_child_data(data_node, "lattice_vectors", lattice_vectors)
    if (.not. allocated(lattice_vectors)) then
      call error("Node 'lattice_vectors' not found or has wrong data type)
    end if
    if (shape(lattice_vectors) /= (3, 3)) then
      call error("Array in node 'lattice_vectors' has invalid shape')
    end if

  end if

  [...]

end select case
```

The lower lying layers warranty that the entire data tree (as sent by the sender) gets trasmitted before the receiver can start to read it. The receiver, therefore, can be sure that it has all the data the sender wanted communicating. It does not need to assume the shape / size of the transmitted data when receiving the message and hope for the best (as it is the case with the bare socket based i-Pi protocol). The arrays in the tree have type and shape information. The receiver can check whether they match its expectations and handle the error gracefully if not.

Debugging communication problems (e.g. sender and receiver implement the high-level protocol differently) should be also straightforward, as the saydx-library contains routines to write the trees from memory to disk.

CHAPTER 2

Comparisons with other transport protocols

There are already several solutions with considerable overlap with the suggested protocol. In order to get a clearer view as to whether it is worth implementing another new protocol, some existing are described and compared against the goals of the suggested SAYDX protocol. In this document, only the transport layer should be discussed here. Possible higher level protocols based on SAYDX (e.g. for exchanging data between atomistic simulation tools) should be described and compared against existing options in a different chapter.

2.1 HDF5

HDF5 is an open-source library (partially BSD, partially MIT licensed) using (and defining) its own data storage format. It is widespread in the scientific community and bindings exist for almost all commonly used programming languages.

SAYDX is probably closest to HDF5 in spirit. The main similarities are:

- Basic building blocks are arrays of data. Type information about elements in an array is stored only once for an array (together with the shape of the array).
- Data is stored in binary format.
- Arrays can be arranged in a tree like structure with named nodes.
- HDF5 allows storage of data (of arbitrary type) at a node. This should be possible in the SAYDX tree as well (although, only as string attributes).

Differences:

- HDF5 is pretty much file I/O oriented. Although it is possible to build up a tree in memory, it is not clear to me (B.A.), whether this tree can then be communicated to a routine in a library, other than being written to disc and being reread again. In environments supporting FIFO, named pipes might be able to partly address this, but this is operating system specific (BH). SAYDX should allow for passing trees between program components (even if written in different languages, e.g. Fortran and C) and running on a range of OSes.
- HDF5 is optimized for handling large amount of data. It had very advanced features, like parallel I/O. On the other hand, it is a very complex library which is not straightforward to build or link. SAYDX should be optimized

for more moderate data amounts (kilobytes to few megabytes, maybe up to a few hundred megabytes). It should have much less features than HDF5 (e.g. no parallel I/O), but hopefully then be much easier to maintain, port, build and to link.

- HDF5 requires the path to each node in the tree to be unique. SAYDX should be more like XML, allowing a node to have several children with the same name (although open for discussions). Using XML-notation to indicate nodes, in SAYDX one could have:

```
<simulation>
  <frame>
    <timestep>1</timestep>
    <coordinates>...</coordinates>
    [...]
  </frame>
  <frame>
    <timestep>2</timestep>
    <coordinates>...</coordinates>
    [...]
  </frame>
</simulation>
```

while in HDF5 (using the same notation) one would have to write

```
<simulation>
  <frame1>
    <coordinates>...</coordinates>
    [...]
  </frame1>
  <frame2>
    <coordinates>...</coordinates>
    [...]
  </frame2>
</simulation>
```

2.2 JSON

The JSON protocol has become very popular in the last few years. It allows a standardized data exchange between different application. Being used a lot in informatics (and also in science), libraries exist for (probably) all programming languages used in science.

Similarities:

- Data can be arranged in a structure, which one may interpret as a tree with named nodes (in JSON they are actually a combination of lists and dictionaries).

Differences:

- JSON was not designed to exchange scientific (numeric) data. Its JavaScript implementation has only one numerical data type (double precision float), although other implementations may handle numerical data differently. SAYDX should offer all important numeric data types (e.g. from half-precision up to quadruple precision – given appropriate compiler support)
- JSON was not designed to exchange array data. Its "array" construct is basically a flat list where each element can be of arbitrary data type. Type information must therefore be stored for each element separately. Shape information (e.g. for a multi-dimensional array) must also be stored separately in the tree.
- JSON was not designed to exchange large amount of numeric data. It is a text based format, so binary data would have to be converted to text first and then back again to binary format.

2.3 MessagePack

MessagePack is a binary protocol to exchange data between applications. Implementations seem to be present in nearly all relevant programming languages, but apparently not Fortran so far. It uses similar concepts to JSON, but is binary based and has a much better handling of numerical data than JSON. It has support for single and double precision float numbers and allows definition of extended types.

Similarities:

- Data can be arranged in a structure, which one may interpreted as a tree with named nodes (actually a combination of lists and dictionaries).
- Data is stored in / serialized to a binary format.

Differences:

- Similar to JSON, MessagePack's array concept is a flat list containing objects of arbitrary types, with the same disadvantages as above.
- As a consequence of the additional type information is stored with each element, hence adding a native (Fortran or C) array to the tree always requires copying and reading out the array from the tree into a native array as well. This is may not be a problem if the tree is communicated via sockets, but could raise efficiency problems if the tree is passed via an API between various components of an application.

2.4 CSlib

CSlib is a client-server library for interchanging data between applications. It allows for exchanging data via files, sockets (via ZeroMQ) or MPI. It is already part of LAMMPS and should also exist as a separate project under GitHub (apparently it has not been uploaded yet). Its licensing is unclear, probably modified BSD, although some documents mention it as GPL-licensed.

Similarities:

- CSlib passes data in binary form. It has data types suited for scientific applications.
- It is possible to transmit multi-dimensional arrays with CSlib.

Differences:

- CSlibs messages are composed of fields, each field being assigned to have an arbitrary data type with zero or more entries of that type. It does not have the concept of a hierarchical tree. However, with an appropriate wrapper, it could be probably used to transmit a tree.
- While it is possible to transmit multi-dimensional arrays with CSlib, it seems that the array shape is not transmitted explicitly (only the number of elements). This shape data would therefore have to be communicated as an extra message.
- CSlib is not designed for passing a tree via an API between parts of an application (e.g. caller passes a tree to a library routine and receives another tree in response), but concentrates on sending it via sockets, file I/O or MPI-messaging.

2.5 MxUI

The MxUI library wraps MPI calls for simplifying Multiple-Program Multiple-Data communication. The library provides a C++ header only implementation. It can also interpolate the transmitted data.

Similarities:

- Data can be arranged in structures, of arbitrary type

Differences:

- Templated push and fetch operations
- Processing (interpolation) of data by transmission