Chapter 2
TDT: A Library for Typed Data Transfer

Ciaron Linstead

2.1 Introduction

Many solutions exist for the problem of model coupling such as a fully inte-
grated coupler like OASIS, geared towards large model coupling problems, or more
basic communication layers like the Common Object Request Broker Architecture
(CORBA), which can nevertheless require a large investment of effort to implement.

Model coupling in climate research, or any other field, comprises several levels.
At it is lowest level, which we address here, model coupling is a technical exercise
in transferring data between two compatible programs. The compatibility of these
programs requires, for example, knowledge of the physical processes being coupled
and the numerical schemes being implemented to model those processes.

The Typed Data Transfer (TDT) library addresses the technical coupling of
modules and provides a means of transferring data between programs in a plat-
form and programming language independent way. This is achieved by means of a
library of functions which can be used in C, Fortran, Python and Java programs, or any
programming language which can import foreign functions from a C library.[1] Hetero-
geneous coupling, i.e. transfer of data between remote machines, is also supported.

The TDT will transfer primitive data types (integer, float, char, etc.) as well as
complex data structures (arrays and matrices, C structures) with one function call.

Non-TDT programs that use the same underlying transfer protocols can also
communicate with TDT programs. TDT's output is simply byte-streams which can

---

[1] All the examples given here will be in C. Examples of the TDT in use in Fortran and Python
are available in the standard TDT distribution.

C. Linstead (✉)
Potsdam Institute for Climate Impact Research,
Potsdam Germany
e-mail: linstead@pik-potsdam.de

be decoded by receiving programs, though at the expense of replicating functionality which already exists in the TDT library.

2.2 Architectural Overview

The TDT has been designed and implemented with several principles in mind:

Ease of Use

The intention from the outset was to design a library which would read or write data with function calls no more complicated than a simple "print" statement. The investment of time and effort by a programmer in simply trying out the TDT is significantly lower than for other, more complex, data transfer or model coupling mechanisms.

Portability

With the desire to share scientific code modules and the trend towards distributed model coupling, it is important that modules be designed to operate on many operating systems and interface with as many programming languages with as little modification as possible. In achieving this, it was decided at an early stage to write the core functionality in C (following the Portable Operating System Interface (POSIX) standard) and use the American National Standards Institute (ANSI) standard for data types. Most major programming languages implement a foreign function interface for C libraries, making the task of using the TDT in, for example, Fortran a matter of writing call-through functions.

Efficiency

The efficiency of data transfers is a major consideration in designing a model coupling scheme, particularly with complex climate systems models. An inefficient mechanism for moving data would be an immediate barrier to its adoption. Thus, the TDT does not perform any marshalling or buffering of data, and no data conversions except byte swapping where necessary. Contiguous blocks of similar data types are sent with one transfer, regardless of whether they belong to the same data structure in the communicating programs. Tests of transfers over raw TCP sockets have shown that there is little difference in speed between TDT transfers and transfers over hand-crafted socket connections.

Flexibility

The TDT is designed such that extensions to its capabilities are relatively straightforward. New data types can be added, as can new fundamental data transfer protocols. The code has been released under the GNU Public License (GPL) so the programmer has complete freedom to extend the library.

Using the TDT from a new programming language is straightforward when the language provides a foreign function interface, although the TDT has been successfully employed in coupling programs written in languages where no foreign function interface exists (e.g. the General Algebraic Modeling System (GAMS)).

Usefulness

In designing the TDT for ease of use another property emerges, that of usefulness. Fitting the "Unix philosophy"—do one thing well—it can be applied to many fields with equal effectiveness and does not require specialist knowledge of a particular subject area in order to be useful.

2.3 Benefits and Limitations of TDT

Transferring data between programs requires that both programs implement code for reading and writing this data. The implementation of this code depends on several things:

- data types
- size of data (e.g. length of arrays)
- the transfer mechanism, or protocol
- byte order, or endianness

Implemented in the traditional way, each program would contain hard-coded information about the data it sends and receives. The TDT aims to minimise this by keeping as much of this information as possible in external descriptions of the data transfer.

Data Types

The implementation of the data transfer code needs to take account the types of the data being transferred. Different architectures may represent the same data type with a different number of bytes and the programmer would normally be expected to know at compile-time how to translate these data types. This adds complexity to the code, and reduces the flexibility with which the coupling can use different data types.

Size of Data

If the size of a data structure being transferred is altered, several changes need to be made to each program. First, the internal structure for storing this data must be changed. Then, the code which handles the sending or receiving of the data needs to be altered to reflect the new size. The TDT library removes the necessity for this second step, by representing the data structure externally in a data description file. This makes program modifications easier and less error prone.

Transfer Mechanism

The underlying protocols supported by the TDT for the transfer of data are: Unix sockets, intermediate files (either on disk or in shared memory), or higher level methods such as the Message Passing Interface (MPI) for transfer between parallel processors. Usually, switching between these protocols requires major modifications to code, with the attendant risk of introducing bugs; with the TDT, the choice of protocol is made by changing a parameter in the modules configuration file. The choice of protocol can thus be changed at various stages in the development process, depending on whether, for example, modules must be distributed (using network sockets) or must communicate very fast (using shared memory).

Byte Swapping

When data is represented in multiple bytes, the order of those bytes in memory is dependent on the architecture of the machine. This is called endianness and has two main alternatives, big endian and little endian. Transferring data between machines with different endianness requires that bytes be swapped at one end of the transfer. The TDT library handles byte swapping internally and is transparent to the developer.

Limitations of TDT

The TDT handles the low-level software infrastructure of data transfers between programs. The semantics of those transfers must make sense at the physical, mathematical and numerical levels to have the coupled model give meaningful results. This is not checked by the TDT, although the data descriptions which the programmer must write can act as basic documentation of the technical interfaces between modules.

The TDT itself performs no data processing, such as interpolation, but one could envisage a scenario where an interpolation "module" is plugged into the transfer using TDT.

Synchronising modules must also be checked manually. Each "read" must be matched by a corresponding "write".

2.4 Model Coupling with TDT

A model coupled with TDT has several elements:

- the modules being coupled
- the data being transferred
- a configuration file per module (XML)
- the description of the data being transferred (XML)
- communication mechanism, e.g. files, sockets etc.

The coupling is a transfer of data between two programs at some point in their execution. This transfer is carried out directly between the components themselves—the TDT does not buffer or marshal the data which would have a detrimental effect on the performance of the transfer.

The timing of the transfer is determined by where in the module code the "read" and "write" function calls are inserted. The matching of read and write operations must be done by the programmer to avoid lost data or deadlock situations.

This transfer occurs over a defined channel which has two principal characteristics: the data it will carry and the type of the underlying transfer mechanism (e.g. sockets, files, shared memory). These two elements are described in two configuration files, the data description file (datadesc) and the configuration file (config).

These files are in eXtensible Markup Language (XML) format, a standard language for metadata markup which has many freely available tools for processing and display.

Each end of the transfer (i.e. the reader and the writer) reads the data description file (or has its own copy of the file in the case of distributed models) which the TDT parses on initialisation. The datadesc can contain descriptions of all data being transferred on a particular channel (multiple datadesc files for multiple channels) or it can describe all data on all channels (one datadesc file for multiple channels).

The datadesc contains declarations of variables, their types and sizes if necessary. A full description of the syntax is given in Sect. 2.5.1.

Calls to underlying system-dependent functions for sending and receiving data are dispatched from the TDT read and write functions (tdt_read() and tdt_write()) according to this datadesc. This is carried out recursively when necessary, for example when transferring multidimensional arrays or C structures.

Each module in the coupled system has its own configuration file. This file defines which input and output channels the program uses, the transfer protocols for these channels, the name of the datadesc file which defines this channels data and any connection-specific information for the channel, for example port numbers and hostnames for socket communications. A description of the syntax used in the config file is given in Sect. 2.5.2.

2.5 TDT in Practice

To use the TDT the following steps, in addition to the normal programming of the module, must be followed:

- create data description and configuration files
- include TDT header file
- declare variables required by TDT
- read the configuration file
- open the communication channel
- perform the read or write
- close the channel

Other, optional, function calls are:

- *resize an array*: this is useful for dynamic arrays whose size is not known when the data description is written
- *send and receive the address of a TDT program*: this is useful when location of one endpoint of the communication is not known in advance, for instance when writing a controller for communication between several modules.

This section includes a simple example of a two-way data transfer between two programs. This example is included in the TDT source code distribution.

2.5.1 The Data Description File

The data description file contains one or more declarations which describe the types of data which will be transferred.

The general format of the datadesc is:

```
< data_desc >

    <decl name="string">
        datatype
    </decl>
    .
    .
</data_desc>
```

where "datatype" is one of the primitive datatypes:

- `int`
- `float`
- `double`
- `char`

or a complex datatype:

- ```
  < struct>
    <decl name="string">
    datatype
    </decl>
    .
    .
    </struct>
  ```

- ```
  < array size="integer">
      datatype
      < /array>
  ```

The "name" attribute of the `<decl>` tag uniquely identifies the data item to the TDT. The calls to `tdt_read()` and `tdt_write()` in the modules use both this name and the variable name to specify to the TDT the data being transferred. In practice, making the name attribute the same as the variable name in the program contributes to good documentation of the interface between the coupled modules. Note that this grammar allows nested arrays. These are handled recursively by the TDT and are transferred with a single function call.

2.5.2 The Configuration File

The TDT configuration files define the channels particular modules will communicate on. The general format is as follows:

```
< program name="string">

  < channel name="string"
            mode=in | out
            type=socket | file
            filename="string"
            host="string"
            port="integer"
            datadesc="string">
  < /channel>

< /program>
```

The <channel> tag has several attributes. These are:

- name: the name of the channel. This is used when opening the channel
- mode: in or out, the direction this data flows
- type: socket, file or mpi, the communication protocol to use
- filename: if type is file, the filename to read from or write to.
- host: if type is socket, the hostname to read from or write to.
- port: if type is socket, the port number to read from or write to.
- datadesc: the name of the file containing the description of the data this channel will transfer.

The configuration file will contain descriptions of all the channels a particular program uses throughout the course of its execution.

2.5.3 Implementation in the Original Codes

Include TDT Header File

Simply add the include directive #include "tdt.h" at the start of the module code.

Declare Variables

Two variables are required for storing TDT specific information, one of type TDTConfig and the other TDTState. One TDTConfig is required per configuration file being read (generally only one) and one TDTState per channel.

Read Configuration File

This step is carried out once per configuration file (i.e. generally once during execution of a program). It reads and parses the configuration file and stores the resulting data structure in the TDTConfig variable previously declared: tc = tdt_configure ("configuration-filename").

Open Communication Channel

A call to the tdt_open() function prepares the communication channel for read and write operations. A typical call to this function looks like this:

    ts = tdt_open (tc, "channel-name");

where ts is the TDTState variable, and tc the TDTConfig variable. The channel identified by channel-name must be described in the configuration file

(i.e. "`configuration-filename`" from the previous step). After this function call, the `TDTState` variable is used to identify an open communication channel. In other words, the `TDTState` uniquely identifies one channel. If data is being transferred to or from more than one location, more `TDTState` variables need to be declared.

Read or Write Data

Calls to `tdt_read()` or `tdt_write()` perform the entire read or write operation for the specified variable. The user specifies the variable to be written (sent) and the identifier (`decl-name` from the `name` of element decl, see Sect. 2.5.1) which appears in the data description file.

```
tdt_read (ts, variable-name, "decl-name");
tdt_write (ts, variable-name, "decl-name");
```

All read operations are blocking. That is, the execution of the program will not proceed past a `tdt_read()` call if there is no data yet available.

Close the Channel

This step closes the named connection, frees up the socket for later use, and frees memory previously allocated for data structures associated with the connection.

```
tdt_close (ts).
```

2.6 Additional Technical Details

The TDT can be used in a number of programming languages. It has two core implementations in C and Python with interface functions supplied for Fortran and Java.

The library has been tested on a number of operating systems, namely Linux (the main development platform), AIX, *BSD, Mac OS X, Windows 98, NT, 2000 and XP and the Cygwin Unix environment for Windows.

Because the TDT has network sockets as one of it's underlying transfer protocols, communication over secure shell (SSH) connections is possible. Thus, for example, transparent communication between remote machines (heterogeneous coupling) is possible.

2.7 Conclusions and Perspectives

The TDT library is a small, self-contained library of functions for the transfer of data between programs written in diverse programming languages and across diverse operating systems. Its strengths lie in three main areas: firstly, its simple interface

using read and write calls is intuitive to apply and requires minimal modifications to existing codes. Secondly, the ease of use of the TDT lends itself to rapid prototyping of coupled models either to try out a particular configuration or to develop an production-ready model. Thirdly, the flexibility of the TDT means that changes to the models being coupled are easily dealt with. For example, different communication channels such as network sockets, intermediate files or shared memory require no changes to existing model code and this means that trade-offs between the distribution of modules (using sockets) and performance (using shared memory) can be explored without major refactoring of software.

Although lacking advanced features such as interpolation, time-stepping and data marshalling, the TDT can be used as a basis for more advanced model coupling techniques. The ease with which the TDT can be employed makes it ideal for rapid prototyping of coupled solutions.

The TDT's user community is made up of several small groups from a variety of backgrounds. These include climate modelling at PIK, geophysical modelling, multi-agent social-biophysical modelling, both in production and for rapid-prototyping of coupled solutions. Also, the Dutch National Institute for Public Health and the Environment (RIVM) has developed the M modelling language for defining and visualising mathematical models. In conjunction with PIK, modifications were made to the M driver software to support communication of these models via TDT. Finally, the Bespoke Framework Generator (BFG) described in Chap. 7 supports the generation of wrappers for models which use TDT as their communication layer.

As an Open Source library of 6,000 lines of source code, released under the GPL it can be easily modified to include datatypes or transfer protocols not originally envisaged by the designers. It is also intended to be applicable to a user-generated suite of model coupling components which will eventually provide a more complex model coupling solution. At present there are no plans to add major new features to the TDT library, though active maintenance is on-going in response to bug reports and user requests for enhancements such as new data types.