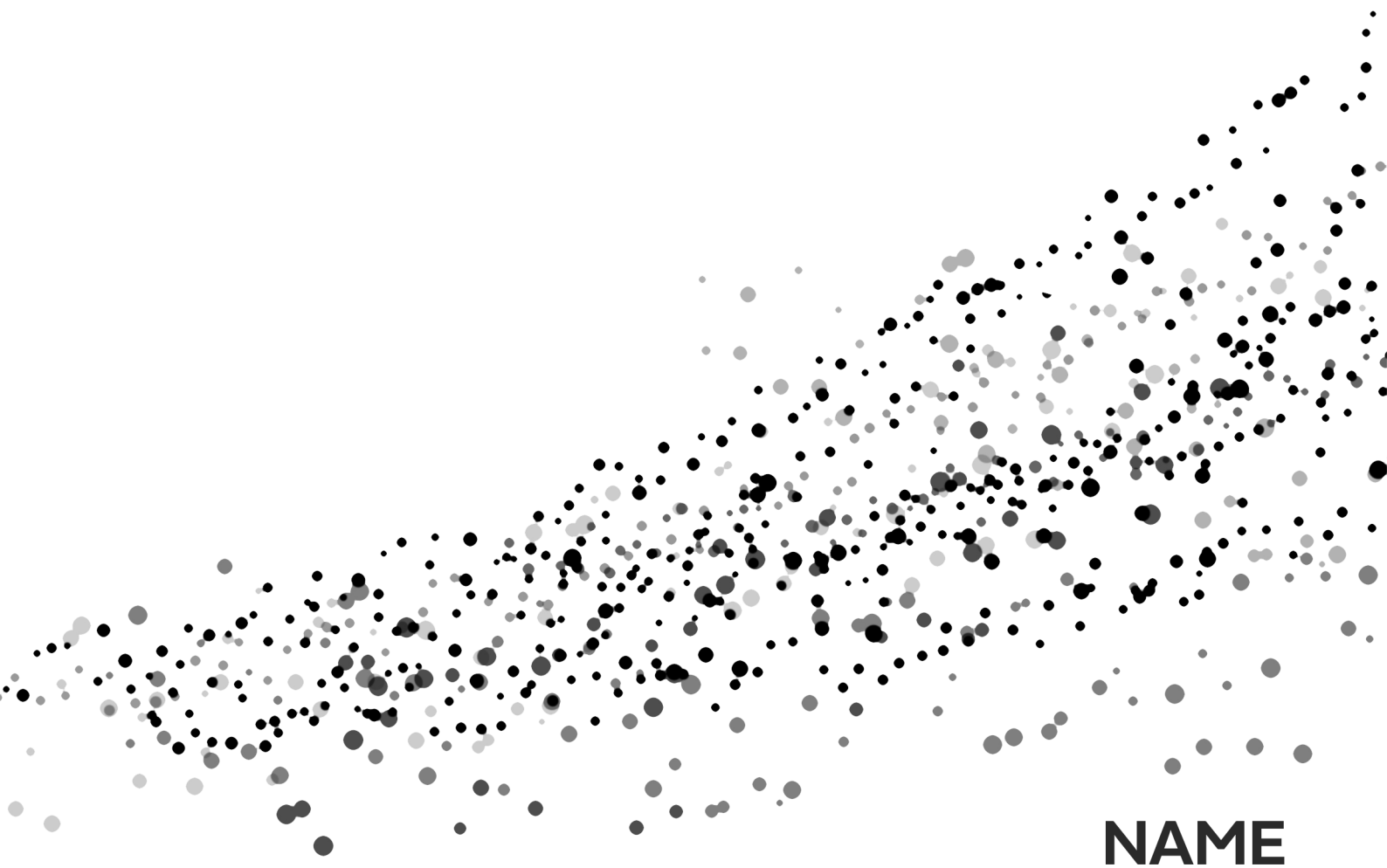
Met Office

NAME Technical Specification Document K03

I/O Server & Supporting MPI Communicator Topology

Matthew Glover

Documentation issued with	:	NAME Version 8.8
Document last updated for	:	NAME Version 8.8
Last updated on	:	20/04/2026



NAME

Numerical Atmospheric-Dispersion Modelling Environment

Met Office, FitzRoy Road, Exeter, Devon, EX1 3PB, United Kingdom

Contents

1 Introduction

NAME requires Met data, which takes time to read. This cost is only expected to rise with increasing data volumes. Moreover, the move towards NetCDF as the preferred format for Met data introduces attendant difficulties around thread safety of the NetCDF library.

The I/O server functionality described in this document takes steps towards the following solutions:

Prefetching: Reading data in parallel with compute work, ahead of the time it is needed. NAME does not currently leverage this, but the I/O server functionality is written to support this in future.

Thread safety: Reading data on a pool of MPI ranks, rather than threads of the same rank. This bypasses threadsafety problems introduced by the NetCDF library, but does rely on the MPI implementation being threadsafe; it requires `MPICH_THREAD_MULTIPLE`.

Handling threadsafety is important for NAME, as OpenMP represents the primary mode of parallelism. This is likely to remain true for the forseeable future, not least because it avoids issues around load balancing between MPI ranks. There are negatives, however. OpenMP does not make hierarchical parallelism easy, particularly compared to MPI. Splitting off some number of threads to perform I/O functions is not a trivial matter: nesting and tasking both bring their own complications, and neither one addresses the NetCDF threadsafety problem.

The MPI picture does require more coding up-front, but it affords much greater flexibility than an OpenMP-only solution could provide. For example, MPI would allow data-reading to take place on a separate node – if that proved to be the most performant way to go – or MPI-3 shared memory window handling would allow specific cores on individual nodes to place data in memory visible to other MPI ranks and their threads on that node. There is also scope to leverage the MPI-IO-based parallelism provided by NetCDF4. The number of dedicated MPI I/O ranks would need to be tuned experimentally, as would the striping characteristics of the Met data.

The content of this document is organised as follows. Section 2 describes the various types of MPI rank and communication patterns between them. Section 3 illustrates the MPI communicator topology required to make this happen.

2 The overall picture

A subset of MPI ranks are ringfenced from the sole purpose of running as I/O server ranks. When NAME is executed, MPI ranks will themselves determine whether they are designated as compute ranks – running the model – or I/O ranks.

Figure 1 illustrates the overall situation. On the right of the Figure, two compute MPI ranks are shown, each with four threads. The left of the Figure depicts single-threaded server ranks, and these lie in a two-dimensional grid.

Ranks marked as “server root” ranks receive messages from threads of the compute ranks. Such ranks represent the root of one I/O server, which may comprise more than one MPI rank. Multiple I/O servers allow multiple data requests to be actioned in parallel; the number of ranks per server allows each individual data request to be parallelised via MPI-IO. In other words, multiple I/O servers can read multiple NetCDF variables simultaneously, but multiple ranks per server can parallelise each individual variable read. The parallelism is two-fold.

Where individual variable reads are parallelised over MPI ranks, each rank in the I/O server only reads a portion of the variable data. There must be some mechanism to put the data into a single array. Currently, this is done with `MPI_Gatherv`; the operation must be able to cope with the load balancing not being exact between MPI ranks in the server.

Moreover, with multiple ranks per server, the striping characteristics of data to be read will likely have a significant performance impact. Some effort would be necessary to tune the stripe count and stripe size accordingly.

At present, there is an additional complication which applies when OpenMP parallelism is already active before the code reaches the I/O loop. In this case a nested parallel region is started with new threads spawned, as illustrated in Figure 2. Underlying use of thread IDs in MPI tags is managed in such a way that multiple nested levels will produce unique tags. This is subject to some limitations on thread count, but these limitations are not likely to be reached; if exceeded, the code will abort in a controlled way.

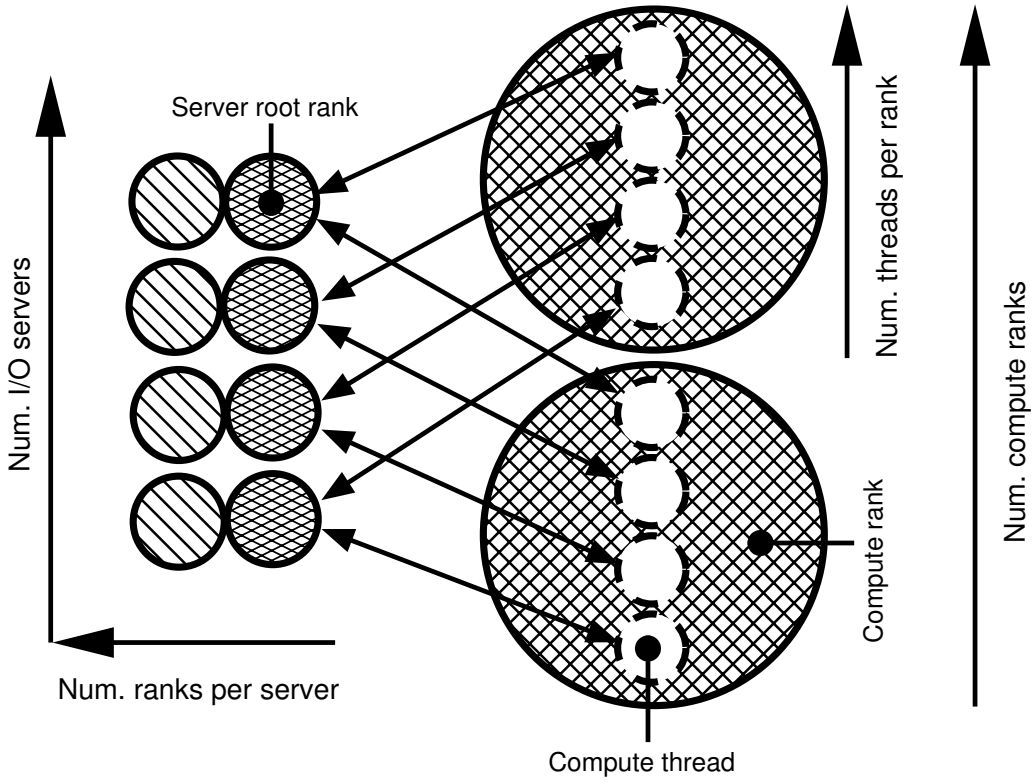


Figure 1: I/O servers in NAME.

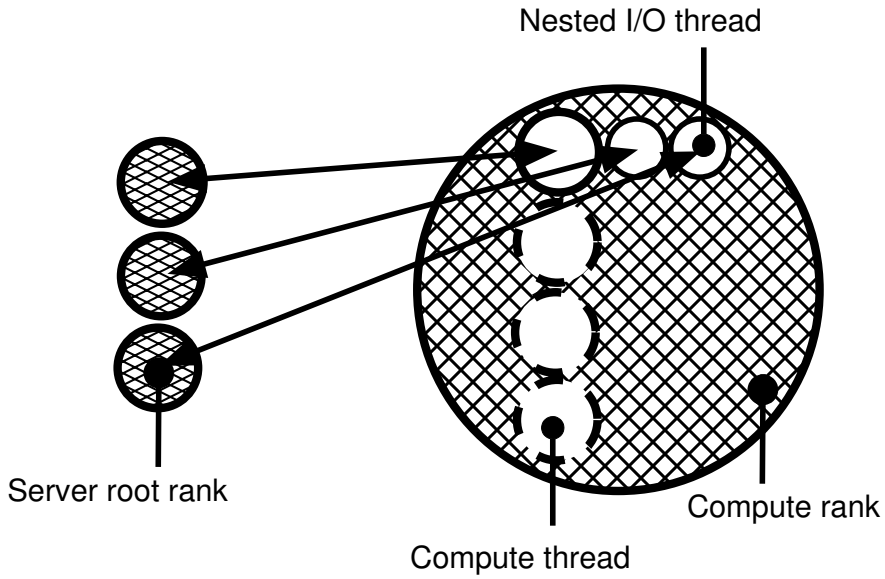


Figure 2: I/O servers called from nested OpenMP.

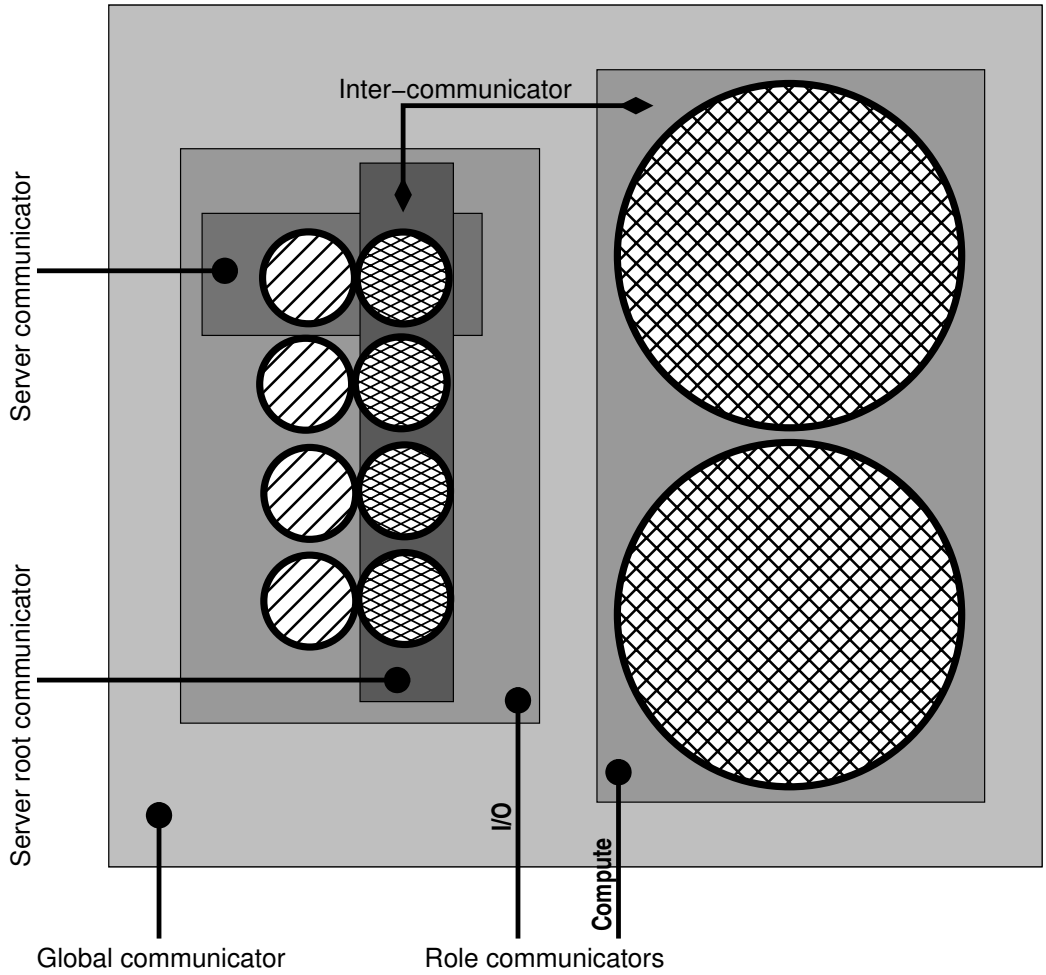


Figure 3: Communicator topology

The nested OpenMP is intended to be a temporary measure, until the implementation of I/O server functionality becomes more mature. It will inevitably have implications for performance tuning, with care needed around affinitisation, thread pooling and OpenMP wait policies.

3 Communicator topology

In order to support the necessary communications between MPI ranks performing different roles, the global MPI communicator is split such that there are separate communicators for compute ranks and I/O ranks. The I/O communicator is divided further.

Figure 3 illustrates the communicator topology, and we describe the various communicators now.

Global communicator: Includes all MPI ranks. This is a duplicate of `MPI_COMM_WORLD`.

Compute communicator: Includes all MPI ranks performing compute work.

I/O communicator: Includes all MPI ranks performing the I/O server function.

Server communicator(s): Each I/O server does not have to be a single rank: there is an additional level of parallelism to accommodate parallel variable reads with MPI-IO underneath NetCDF4/HDF5. Each server communicator includes MPI ranks comprising one I/O server. Where there is only one MPI rank per server, the number of server communicators will match the number of I/O servers, each with size 1.

Server root communicator: Includes the root ranks of each I/O server.

Inter-communicator: Allows any compute rank to communicate directly with any rank in the server root communicator. When a thread of a compute rank requests data from the I/O server, it addresses the specific

server root by its rank in the server root communicator.

4 Tuneable parameters

There are a number of environment variables that control the configuration of I/O server ranks, and also a number of parameters used to dimension arrays internally inside NAME.

4.1 Number of servers & ranks per server

To activate basic I/O server functionality, only `NAME_NUM_IO_SERVERS` need be set. There is additional complexity around launching the MPI job and MPI rank affinitisation, but the other environment variables need not be set unless the defaults prove unsuitable for a particular configuration of NAME.

NAME_NUM_IO_SERVERS: The number of I/O servers, which corresponds to the number of server root ranks when there is more than one MPI rank per server. Default: 0.

NAME_NUM_RANKS_PER_SERVER: The number of MPI ranks per I/O server. This variable should only be used in conjunction with NetCDF4 parallel support. Default: 1.

4.2 Other parameters

The following variables are used to dimension arrays internally inside NAME. They carry sensible defaults. It is not necessarily a requirement for any of these to be set by the user.

NAME_SERVER_MAX_BUFFERS: The number of data buffers on each I/O server rank. Individual buffers on the server cannot be reused until data has been sent to the compute rank that requested it. Multiple buffers allow multiple reads from disk to take place before communication of that data back to compute ranks has completed. If there any time is spent waiting for buffers to become available, this will show up in the server output as “stall time”. Default: 20.

NAME_SERVER_MAX_OPEN_FILES: The number of files that will be held open simultaneously by the I/O server ranks. If data is needed from a new file, and the maximum is exceeded, the I/O servers will close the least recently opened file. Default: 3.

NAME_MAX_READ_REQUESTS: The number of read requests, made by compute ranks, that can be tracked by compute ranks. If the number of requests exceeds this number, NAME will abort and advise the user to increase its value. With the code as it stands today, there is no prefetching; there will only ever be a maximum of one outstanding request on each compute thread. Default: 1.

5 Making I/O requests

At the point that data requests from the I/O server are made, the Fortran code looks the same regardless of whether or not the I/O server is in use. The “strategy” pattern – from object oriented code design – is used to determine what happens at these calls. The NAME module requesting the data declares a polymorphic “batch reader” object.

```
class(batchReader_), allocatable :: readerStrategy
```

Subsequent typed allocation occurs, which fixes the type of `readerStrategy` and hence determines the code that executes at those calls. The fixed type – or *concrete* type – is either `readerMPI_` or `readerSerial_`, depending on whether or not I/O servers are active.

```
!Allocate module-level variable.
!$OMP PARALLEL
if(.not. allocated(readerStrategy)) then
  if (serverActive) then
    allocate(readerStrategy, source=readerMPI_())
  else
    allocate(readerStrategy, source=readerSerial_())
```

```
    end if
  end if
  !$OMP END PARALLEL
```

Since the `readerStrategy` objects are private to each thread, this allocation must happen inside an OpenMP parallel region. Note that this allocation needs to happen once only, at the start of a run.

Once the setup process has occurred, then the calls required to request data are the same in both the parallel and the serial case:

```
  call readerStrategy%add_request()
  call readerStrategy%finalize_requests()
```

The second of these calls is blocking: the code will wait until all requests have completed before moving on. With the code as it stands today, without prefetching, one request is added and waited for immediately; longer term, these calls would be separated in the code, allowing multiple requests to be added long before a call to `finalize_requests()`. In this way, Met data would be read ahead of the time it is needed by I/O server ranks.

6 Launching NAME

Although the same NAME executable is launched for compute and I/O ranks, jobs are launched as Multiple Program, Multiple Data (MPMD). The need for this stems from the different threading and attendant resource requirements of compute and I/O server ranks. I/O server ranks are not multi-threaded, requiring only one thread and one core; compute ranks will typically run with many threads, requiring many cores.

Launchers – typically `mpiexec` – vary from machine-to-machine. The precise command line arguments required to launch NAME will depend on the specific launcher options.

7 Serial performance of NetCDF

We have observed significant performance issues relating to how compute ranks and I/O ranks are laid out within supercomputer nodes.

The issues seem to originate from decompression of NetCDF data. For this reason, NetCDF data used today for testing is not compressed. Having experimented with NetCDF compression levels, any degree of compression incurs a large performance penalty; the penalty is exacerbated enormously by resource-sharing between compute ranks and I/O server ranks. It can matter whether threads of compute threads – even “pooled” threads – share the same socket, or sometimes even the same chiplet, as I/O server ranks. Whether or not compute ranks carry an active or passive OpenMP wait policy can have a huge impact on read times by I/O server ranks; moreover, in measurements to date, neither wait policy was universally better than the other in this regard.

It is these findings which motivated the decision to place compute ranks first in the global communicator, with higher-numbered ranks being I/O server ranks. This proved to be the least problematic configuration when compute threads carry an active wait policy; this policy is usually desirable for performance reasons when there is only one thread per core (no Simultaneous Multi-Threading – SMT – or Hyperthreading). Changing the MPI rank order, say through MPICH environment variables, may carry significant performance risks.

Although uncompressed data is reasonable for testing in the short-term, with today’s data volumes, we see this as a temporary workaround and not a solution. Further investigation is needed.

8 Future development

As raised in earlier sections, there is a longer-term aim to enable prefetching with the I/O server. Indeed, that was the original reason behind writing an I/O server; the thread safety aspects are a useful by-product.

The I/O server code itself, as it stands today, is simple; it includes just enough complexity to give performance gains compared with serial code. The I/O server ranks do not use threading, as we did not see an immediate need.

In order to get the prefetching working as intended, some additional complexity *might* be required so that data is fetched back onto compute ranks completely asynchronously, without waiting for the call to `finalize_requests()` outlined in Section 5. For example, it is in the nature of asynchronous MPI communications that data from I/O server ranks sent to compute ranks might not reach their destination if no calls are made to MPI on the compute side. If compute ranks request data, but then become engaged in a long-running piece of compute work, then data may not have reached buffers on the compute side. In other words, data would not yet be in-place on compute ranks. It is possible, however, that it would have been read and buffered by I/O server ranks provided that there were enough buffers available (see `NAME_SERVER_MAX_BUFFERS` in Section 4.2). The MPICH implementation of MPI provides an environment variable – `MPICH_ASYNC_PROGRESS` – which spawns a dedicated thread to handle asynchronous data movement.

9 Summary

NAME can read data from NetCDF files on dedicated I/O server ranks. Multiple servers can read different variables from the same file in parallel. A second dimension of parallelism allows parallel reading of individual variables, via MPI-IO support underneath NetCDF4.

The original intention behind separate server ranks was to allow prefetching, where Met data would be read ahead of the time that it is needed. While this remains a key aim, it is not enabled at present. The major benefit in the first instance – with the code as it stands today – derives from parallel reading of many variables, made possible by side-stepping thread safety issues in the NetCDF library.

In order to support this functionality, there is a hierarchical communicator topology, with different communicators for MPI ranks performing different functions; for example, a given MPI rank might be a compute rank or an I/O server rank. When the I/O server functionality is used – when `NAME_NUM_IO_SERVERS > 0` – it becomes necessary to launch the NAME executable as if it were Multiple Program, Multiple data (MPMD). This is to accommodate different thread count requirements of compute ranks versus I/O ranks.

Performance measurements to date have revealed subtleties around how compute and I/O server ranks are laid out within a node, with potentially huge variations in performance depending on how this is done. Such issues are largely avoided if data in NetCDF files remains uncompressed. While this provides a tolerable workaround today, it is not a long-term solution. It needs further investigation.