Deliverable 12.2

EISCAT_3D System Control Report

Jussi Markkanen and Assar Westman

February-2014

Table of Contents

1 Introduction

In our 2009 EISCAT_3D Design Study report on control and monitoring subsystem (henceforth, [DSR2009]) we argued that un upgraded version of the current EISCAT system control software, Eros5, should be the next step on the path towards a base-line E3D control software.

Two observations form the basis of this "an Eros upgrade only" -argument. First, from the user and usage point of view, EISCAT 3D must be an integral part of EISCAT overall system, rather than something drastically different. Therefore, uniform control software should preferably be used across all EISCAT systems. Second, over 30 years Eros has proved to be a reasonably flexible, steadily evolving, system for conducting measurements on the multi-site EISCAT radar. We argued that starting E3D control software development from scratch would entail a serious risk of failure, for little obvious gain.

Back in 2009–2010 there were some minor though nagging problems with Eros5, and some aspects that were seen to play prominently in E3D where hardly available in Eros5 at all.

Reflecting these premises, the tasks of the Preparatory Phase Work Package 12 study were designed to facilitate the next evolutionary steps in EISCAT system control, first the move from Eros5 to Eros6, which would still be mainly apply for the present EISCAT systems, and then later to Eros7, which would include day-one support for E3D. The specified tasks were the following.

(12.1 Generalization of Eros to multiple nodes. *This task was cancelled.*)
12.2 Extension of Eros to allow novel radar operations.
12.3 Automation of system state messaging.
12.4 Initial implementation of external programmatic Eros control.

In his report we explain what we have done and not done to implement these steps. After this Introduction, the report consists of two major parts; then a brief concluding section; and finally a lengthy Appendix. The first major part, Chapter 2 of the report, is organized along the official Task structure of the WP, and is still somewhat introductory in nature. During the more than three years in the work, the ideas have evolved. With our current hindsight, we re-visit the background reasoning of the Task specifications. We explain why we have ended emphasizing some Tasks more than others, why we have ignored some tasks almost completely, and how we have, de facto, added a new task, concerning the overall structure of Eros.

The second major part of the report, Chapters 3 to 5, is more technical in nature, and is organized along the natural logic of the actual Eros upgrading work. We describe, often on code level, our solutions to some of the key problems in the upgrade. However, we are *not* attempting to make this report a user manual of Eros6. Such a manual is needed, and will appear in due time, but here we are concerned about those Eros internal matters that most Eros end-users will never see, and should never need to worry about.

Many of the solutions make use of new features of a recent major upgrade of the Tcl/Tk programming language with which much of Eros is coded. Especially, we make strong use of the programming construct called the *coroutine*. Coroutines are still so new in Tcl programming that very little material about them is available from standard Tcl information sources. We give a short general introduction to Tcl coroutines in Section 3.3. One of our general coroutine-based solutions to inter-process communication (IPC) we

have not yet found published elsewhere. Chapters from 3 to 5 assume basic knowledge of Eros and Tcl.

After the three technical chapters, we give a summary of the work in Chapter 6, and then in the concluding Chapter 7 outline what needs to be done next in the control system software development. An Appendix explains how Eros6 utilizes certain internal properties of the standard Tcl IPC package COMM, to allow communications between some Eros nodes whose full network address is not readily available.

2 The work package Tasks

2.1 (Task 12.1 Generalization of Eros to multiple nodes)

Task 12.1 was deemed to be premature without actually having the multiple nodes available, and was dropped in mid-term. Nevertheless, mainly as an application of the software developed in the WP, we have performed some simple multi-node tests. They are described briefly in Chapter 5.

2.2 Task 12.2 Extension of Eros to allow novel radar operations

Task 12.2 is concerned about Eros commands required for the E3D system, on top what is available already for present EISCAT systems. There will be novel hardware at E3D. This undoubtedly will require new Eros commands to be created. But the final E3D hardware specifications are still taking form when this report is being written. We do not know how to design hardware commands without the hardware or at least the hardware specifications at hand. We have also argued in [DSR2009] that once the target hardware is known, interfacing it to Eros would mostly be straightforward and fast. No new hardware commands have been added in the present work. This will be the main task when upgrading Eros6 to Eros7, once the E3D hardware becomes available.

In this WP, quite a few new Eros commands, or enhancements to old ones, have been added over and above Eros5, both on user level and behind the scenes. These commands are not directly related to hardware control, but rather to generic aspects of radar operations such as notifications and multitasking. Such features are desirable also for the present EISCAT systems, and, importantly, can be properly tested on the present systems. It therefore makes good sense to develop them at this time already. These new commands form one half of the upgrade from Eros5 to Eros6. The other half is change in the overall Eros architecture, which we will describe in the last section of this Chapter. Some of the new commands are used to implement the Tasks 12.3 and 12.4 of this WP, while some others enhance the communications facilities of Eros and in a sense could go under the cancelled Task 12.1.

2.3 Task 12.3 Automation of system state messaging

Task 12.3 concerns asynchronous notifications within Eros. Eros commands have always been able to report back when something goes wrong or needs attention when they are executed, but that has mostly been done synchronously. A user command is sent out to an Eros subsystem and then the calling program waits (“blocks”) until the command completes in the subsystem, either with a return status of success, or with some kind of error status. While remote call blocking has not been much of a problem so far in Eros, it can become wasteful in the multi-user, multitasking, fast context-switching environment envisioned for E3D.

Most commands in Eros in practice succeed almost always, and do not actually need to return any value for the caller. Also, they are often independent so that they do not need to be given in any particular order; typically this happens at experiment start-up. In

such situations it makes sense and speeds things up for everybody if one just throws in commands as fast as one can without waiting them to complete. But then one has to trust that the target subsystems complain, notify, loudly enough and soon enough in those rare cases when something bad actually happens. The recipient of the notification can then for example abort everything and start from the beginning.

In this work package we have developed a system that allows the subsystems to pop-up a window in the appropriate place and sent messages to that window. Optionally it is possible to request the system to speak-out those messages, too. The scheme relies there being an operator around to observe the notification messages. With painfully loud alarms, this kind of scenario has worked well with EISCAT Tromsø transmitter error handling. The operator-centred scheme can be enhanced, by allowing the subsystems trigger actions that abort an executing script automatically when the situation is classified hopeless enough.

2.4 Task 12.4 Initial implementation of external programmatic Eros control

Task 12.4 handles context switching and multitasking in Eros. By far the biggest developing effort in the WP has been devoted to this task. The tittle notwithstanding, we will not actually *implement* any control here. Rather, we provide facilities that make such implementation useful. The basic capability to use external programs to control Eros has been available and in use for a long time, via the so-called ECO interface. We will briefly mention ECO again in Section 2.5.2.

We have explained in [DSR2009] that context switching in Eros means changing the state of the Eros top-level "state machine" (Figure 5 in [SDR2009]), by either stopping or starting an experiment, or, typically, using an external GOTOBLOCK command to break out from an infinite loop in one block and to transfer execution to another block in the experiment script. It is envisioned that for E3D the context switching by external commands is used even more often than in present systems. It is imperative that such a switching works fast and robustly.

In our [DSR2009] Section 7 we stated that during the Study, "*[...] the GOTO command [...] was implemented in a proper way*". But soon afterwards several EISCAT-users, including us, observed that the context switching in Eros5 still was not reliable. We re-addressed the situation in this WP. We found a genuine bug in the Eros5 implementation of the GOTO command, which causes the command to fail systematically in some rare circumstances. Normally those circumstances are not met, but nevertheless the context switching, typically at an experiment stop, sometimes fails. In this WP we provide a novel implementation of the context switching, based on the new coroutine feature of the Tcl programming language that Eros is written with. The new feature only became available at the end of 2012 when a new Tcl major version, version 8.6, became public. After the unfortunate statement of "proper implementation" of GOTO, we are not going to make any claims about ultimate correctness this time. With the testing done so far, we have not observed any failure in context switching.

2.4.1 Multitasking

The coroutines have allowed us to introduce user-level multitasking in Eros. Eros itself has always been a multiprocessing system via its use of multiple Unix processes. Coroutines instead allow one to implement co-operative multitasking within a *single* process, like the EXPDOER process that Eros5 uses to execute an experiment script. With this kind of user-level multitasking, one can for instance perform several commands in parallel in an experiment script. Or one can run an experiment in one task and maintain an automatically updating monitor window via another task. Or, if it will one day make some sense, one could run many experiments simultaneously within a single "EXPDOER".

Experiment-level multitasking via coroutines is a major new feature in Eros6, and will be described in Chapter 3 of this report. We will provide a fairly detailed description about its implementation. The coroutine feature is so new in Tcl that, as far as we know, none of the standard books about Tcl programming not even mentions it, much less says anything about how to make best use of it. So we have read, and then re-read, the Tcl coroutine manual page and the Tcl coroutine wiki, which both are rather terse.

Besides context switching and multitasking, the third main item we have spent a lot of effort in this WP is how to handle long-living commands. It is unclear under which, if any, of the Tasks 12.2-4 that precisely belongs; but it surely belongs under Eros6. Long-living commands take a long time to complete. In the present EISCAT Heating system, for instance, there are commands that can take ten seconds to complete.

When a long-living command is done synchronously in a non-multitasking system like Eros5, it blocks the calling program for that time. With multitasking, it might be possible to do something else useful during that time. Though, waiting for commands to get ready cannot always be avoided even on a multi-tasking system. If a command produces output that is required as an input for another command, one has no options other than to wait for the command to become ready before issuing the other command.

Referring to Figure 1, which is reproduced from Figure 4 of [DSR2009], the problem that the long-living commands pose in Eros, and especially in the future multi-user, multi-experiment E3D Eros, is not on the client side (EXPDOER process). The real problem is on the server side (the ENGINE process), where the commands are sent from EXPDOER to be further propagated over to the appropriate hardware subsystems. While the client might be motivated to wait for his command even for a long time, the server should not do any waiting that is not absolutely necessary.

Eros has always supported server-level multitasking via the Tcl event loop. In Eros5, prior Tcl 8.6, the multitasking is implemented using so called VWAIT mechanism. It is well known in the Tcl community that this can lead to unnecessary delays (and worse) in some situations. This has meant that in Eros5, the server has been living dangerously. A single long-living command can block the progress of short-lived commands for the whole duration of the long-living commands. We will show examples of this in Chapter 4.

For a multi-user system, that kind of behaviour should definitely be avoided. In this WP, with the help of the new Tcl 8.6 and some advanced features of the standard Tcl inter-process communication package COMM, we have been able to implement a good solution for handling long-living commands on servers.

2.4.2 Temporary switch of an experiment

The detailed specification of Task 12.4 raises the question about what should happen when an on-going experiment needs to be suspended temporarily for the duration of some short-lived special event. Could Eros somehow return back to the original experiment and continue as if nothing had happened?

We have not really solved this problem in this work. The coroutine mechanisms allows one to suspend the processing of an experiment script for any amount of time, and, with the multiprocessing facilities of Eros6, another experiment could be launched without trouble, and that could then be stopped in the normal way when it has done its job. But what should happen then? Typically, the interrupting experiment would have changed the radar hardware state in some way, and it would in general make little sense just to continue executing the suspended script from where it was left. One also needs to note

that it would normally not be sufficient just to stop the high-level experiment script from issuing new commands, but one would also need stop the hardware subsystems (in the present EISCAT, the radar controllers) from proceeding with their micro-programs. Continuing the script execution would make sense only if the radar hardware could be brought to the state where it was when the interruption occurred.

To arrange for this efficiently appears to require a drastic change in how experiments have been designed and implemented in EISCAT so far. Even when we have loosely talked about an EISCAT experiment running as a state machine, that refers more to the software on the top level (the experiment file subroutine level), than to the state where the radar hardware is at any given time. The basic problem is that in EISCAT, the "hardware state" is created incrementally, piece by piece, over time. The hardware state at any given moment during an experiment depends on the experiment start-up state, which Eros tries to arrange to always be the same one, and all the commands done in the radar system up to that time.

Within the current Eros practice, the best we can do to specify the hardware state in software is the (implicitly known) initial state, together with all the commands given after that. The other possibility to specify the state, never attempted in EISCAT so far, would be to have some kind of large hardware State Table, where each microsecond would have its own entry which completely specifies the system state for that instant of time.

The present Eros would not know what to do with such State Table. Presently, for better or worse, Eros experiments are not micro-programs, and Eros is not a micro-controller. Maybe at E3D such a brute-force approach to hardware control could eventually become practical. At least conceptually, the overall control program could be some kind of micro-program, where each line fully specifies the system state. Many if not all of E3D subsystems will be controlled by some kind of micro-programs anyway.

But disregarding the idea of a complete redesign of Eros as unrealistic for the time being, are there any other options? It seems that the only way to reliably re-create the system state at some given moment would be re-run the experiment, maybe in some kind of fast-forward mode, from the initial start-up (reset) state until the desired instant of time. In some degree Eros5 already does that when it uses what is called the skipping mode to restart a stopped experiment with its original start time at some site, in order to synchronize with on-going experiments at other sites. A complete solution for the state re-creation would need to take into account also all interactive context switches and other interactive system commands.

Re-creating of the overall system state would become more practical if each subsystem would have micro-program like state, so that the last command given to that subsystem would always uniquely specify the state. Then one would need only to keep track of the latest commands given to the subsystems in order to be able to re-create the radar state. The present EISCAT has subsystems that have this simplifying property, and subsystems that do not. The UHF antenna pointing direction only depends on the latest given command. The state of the radar controller can depend on history, because it is possible to load the radar controller piece by piece.

Re-creating the hardware state given the complete command history does not seem outright impossible, especially if some restrictions can be placed. But it surely is not straightforward either. What if there are many experimenters running on the radar, and only one needs to be stopped and re-created? Could one, for instance, re-enact all the possible hundreds of E3D antenna group firmware module loadings that might have

taken place, in a way that only the experiment that was interrupted is affected? Probably some more powerful ideas are needed here.

It anyway is clear that full, systematic, machine readable logging of all commands, both from scripts and interactive, that change the system hardware state in any essential way, is desirable, for all kind of reasons. This task has been somewhat neglected in Eros5. In Eros6, it goes straight to the system kernel.

2.5 Task 12.1-4 New Eros architecture

2.5.1 EOS and ESH

An essential part of the work done under this WP is a redesign of the overall Eros architecture. The design work overlaps all the WPs tasks, including the cancelled Task 12.1.

We basically view an EISCAT radar as a signal-processing factory. There are a smaller or larger number of functional subsystems forming the signal path to generate, receive, process and store the signal; a timing layer for microsecond-scale timing and synchronization; and the Eros system for the overall, one-second timescale, control.

The present operational Eros, Eros5, as depicted in Figure 1, reproduced from our Design Study report, is a somewhat monolithic system.

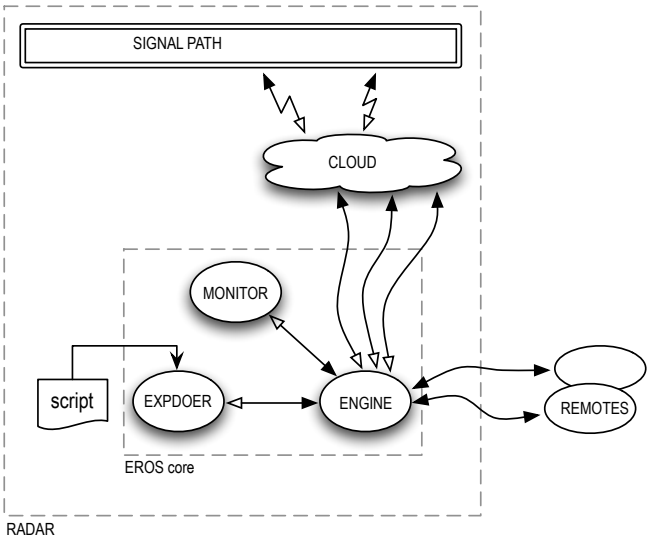

Figure 1. Eros5 top-level structure. *Eros5 is implemented as a core of three interacting Tcl/Tk interpreter processes, all executing in a single Unix workstation, and a radar-site-depend "cloud" of other programs and processes living in many type of computers available to the core via the local area network. (Reproduced from Figure 4 of [DSR2009].)*

Even the core of Eros5 is monolithic, handling both hardware access and running of experiment scripts.

Eros6 provides cleaner separation between hardware access and control, and conducting measurements and experiments with that hardware. In Eros6, the control software is split into two parts, the e-shell ESH, and the E-operating system EOS. In terms of Figure 1, the EXPDOER becomes the ESH (or rather, an ESH, one of maybe many), and is moved out of the Eros core. The new Eros architecture is sketched in Figure 2.

In fact, the EXPDOER/ESH is moved out the box labelled RADAR in Figure 1 altogether, and significantly demoted in status. In Eros6, the programs used to execute experiment scripts are considered to be mere applications, rather than an integral part of the radar operating system proper. The EOS lives at the radar site, in the local net there. But with the high-speed internet, there are no compelling reasons why programs that run experiment scripts should live in the radar LAN.

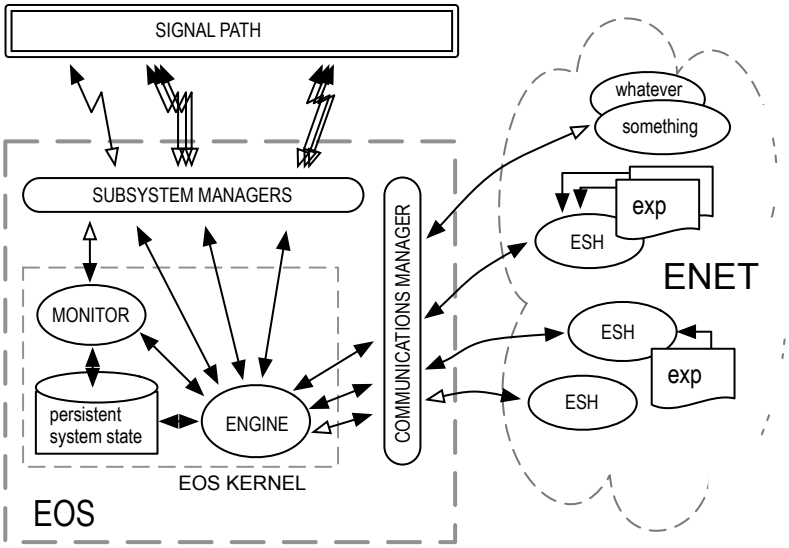

Figure 2. A plan for top-level Eros architecture beyond Eros5. *After Eros version 5, the software's functionality has been split in two main pars, the E-operating system EOS and the e-shell **ESH**. Each EISCAT site has its own local copy of EOS living on the LAN. The EOS consists of a kernel, identical at all sites; a common communications support module; and site-specific subsystem management modules. EOS is the primary software interface to the radar hardware units and signal processing systems. This interface is provided in the form of a set of control and monitoring functions, the Eros "system commands". The main task of EOS is to serve Eros system commands as requested by net clients. In addition, EOS must keep some kind of abstraction of the radar system state, but how that should best be done is not clear. In this WP, we suggest that the state could be represented as a system command history database, which will allow the re-creation the hardware state for any given moment of recent past. The third task of EOS is to perform certain administrative services, like keeping track of which experiments are running by whom using which resources on the radar. Differently from earlier version of Eros, the core system does not run any experiments. In itself, EOS does not even provide an interactive system console, but an ESH can be used for that purpose. EOS only serves, and logs, the incoming stream of system commands. It is ultimately up to the users how the command streams are generated. The e-shell, provided in this WP, is one possibility. It forms a Tcl/Tk-based command-line interface for performing system commands on EOS. The script-processing facilities of ESH are downwards compatible with Eros5, so that existing .elan files can be executed without modifications in ESH. An ESH process can be running in principle at any location on the internet which has secure access to the radar LAN. That part of internet is labeled "ENET" in the Figure. In the minimum, ENET coves the EISCAT VPN.*

The EOS versus ESH split of Eros is partly just semantics. All the old Eros5 software, and quite a lot new, is still somewhere there. But we think it is good semantics. First, it allows us to stick to the convenient notation, when we describe how things work, that a radar's operating system is co-located with the radar hardware. Second, the split suggests that there are, or will be, more choice of the radar user interface than just a single program. We are planning to provide a Python version of ESH, including most of the multitasking support, in a not too distant future. We will also provide an interface from Unix shells to EOS. We have already done this for Eros5, in the form of the ECO Unix command-line program; that works with Eros6 as is, but will be streamlined further.

Also, those presumably sophisticated programs that will decide when to switch experiments in E3D based on geophysical conditions, can be coded in whatever language, and be running anywhere where they have access to their required input, and then just use the string messaging to request appropriate actions from one or more EOSs. In the simplest implementation, those programs could execute the required EOS commands by invoking ECO.

2.5.2 Benefits of the new architecture

Two main benefits follow from this division of work. First, it simplifies the EOS kernel. The kernel needs be able to handle a stream of system commands and maintain the system state. It will also need to keep some kind of record of who is doing what on the radar for administrative purposes. But it needs not care about looping, synchronization, or any other control of script execution. Especially, EOS does not need to manage the experiment context switch that has proved such a problem in Eros5.

In fact, except for administrative and accounting purposes, EOS would not even need to know about the concept of an experiment. It would only need to serve the individual system commands that are sent to it from any number of ESHes from all over the net. But it is practical to assign also certain administrative tasks to EOS. This is partly because EOS, unlike the ESHes, has a fixed, "public", net address. The Appendix of this report describes in excruciating detail how access through EOS is used to allow an ESH2 to talk to an ESH1, in order to stop on experiment running in ESH1, without ESH2 actually knowing the net address of ESH1.

The other major benefit of the Eros split is that there can be any number of ESH peers around, doing any number of things. Some ESHes might act as EOS consoles. Some ESHes might be monitoring, programmatically or interactively, experiments running in other ESHes. Other ESHes might be running experiment scripts, or some other programs. This can happen anywhere in the net, access speed and authority allowing.

An important corollary of the EOS/ESH separation is that the actual radar operating system, the EOS, can be made more robust and secure. In Eros5, there are not many limitations on what kind of commands and scripts users may feed to it. In Eros6, EOS kernel knows that it only needs to handle a rather limited set of stand-alone commands from the outside world, the system commands. Those can be carefully checked before they are executed, and everything else can be disregarded. This reduces considerably the risk that a user inadvertently does something catastrophic on the multiuser system. In the same vain, on the ESH side one can be more relaxed. If an ESH crashes due to an internal error or due to creative programming by the user, nothing very bad can result to other users of the radar.

3 The e-shell ESH

3.1 The ESH program

The ESH program is basically an enhanced Tcl/Tk WISH executable. Normally, ESH is running in an interactive mode, but it can also be started in a non-interactive mode. For the latter use, see Chapter 5 about ESHPOOLs. Tcl/Tk 8.6 is required, but the plain WISH of the standard Tcl/Tk distribution is sufficient, the EISCAT-enhanced version of WISH (WISHE) that is required on the EOS side, is not required here. ESH is coded in pure Tcl, so should work in any system that supports Tcl/Tk 8.6 and the Tcl standard library TCLLIB. When ESH is started, it requires at least one parameter, the user ID (*uid*). Typically, that would

be the initials of the person starting the program. It is up to the user whether each ESH in his computer uses the some or a different *uid*. When an ESH communicates with radar EOSes or peer ESHes, its *uid* is automatically propagated to the target system to identify its remote user on a voluntary basis. (The contents of the *uid* tag are not checked, though the idea is that the tag would uniquely identify an EISCAT user.)

An ESH communicates with EOS using a single basic command (in two variants), implemented with the Tcl COMM inter-process communication package. That package in turn is built on top of a very simple, well defined and document socket communication protocol, the Tcl "wire" protocol. Any programming system, say a Python interpreter, that can write and read text messages to and from socket channels, should be able to talk with EOS, and hence the radar hardware, in a manner similar to ESH.

Contrary to the radar EOS, an ESH does not have a fixed network address. It is possible to use the *–p port* start-up option to specify the port number where ESH listens to incoming connections. By default ESH lets the operating system to assign the port number automatically. Every ESH in a given computer will have a unique port number. Taking into account the IP address of the computer hosting the ESH, the two-element list "*portnumber IPaddress*" identifies a particular ESH.

If the *–p* option is not used and the port number not made public by some external means, the question arises, how can other users contact a particular ESH if they need to do so. Probably some kind of explicit registration to some central server (at the E3D core site) will ultimately be needed for administrative and security reasons. (For security, the communication channel itself probably needs to secured using SSL or ssh tunnelling.) From technical point of view, that is not strictly necessary. The COMM package is powerful enough to keep track about who is in contact with whom. That information can be made available to all interested parties on some fixed, well-known place in the net.

A typical case when this becomes an issue is when a user starts an experiment running in his local ESH, ESH1, accessing radar hardware at EOS1, but someone else wants to stop the experiment using an ESH2 somewhere else in the net. Using the properties of COMM, Eros6 arranges things so that when ESH1 starts the experiment, the start command RUNEXPERIMENT registers the started experiment with EOS1, and at the same time, registers information about how to access ESH1 via EOS1. Specifically, an identifier of the EOS1 side of the bi-directional ESH1–EOS1 communication channel is saved. ESH2 can then find out which ESH started the experiment and how to contact it by executing the Eros PRINTEXPERIMENT *expname* command at EOS1. Details about how this works are given in the Appendix.

We also show in the Appendix that this is the best that can be done without some kind of central registry. In general, to access other ESHes, ESH users might need to go via a radar EOS. The fully qualified net address of the ESH1 is not always available programmatically, not even to the EOS1 COMM, as there might be various address translations, tunnelling, etc., between ESH1 and EOS1.

In order for outsiders such as ESH2 to stop the experiment running on ESH1, the communication channel from ESH1 to EOS1 must be still open when the stop is required. If the communication for any reason is lost, it probably is best to abort the experiment at both ends of the ESH1↔EOS1 channel. COMM detects when a channel is lost, and traces can be hooked to that event in order to stop the experiment automatically.

3.2 ESH remote commands

ESH presently has two user-level commands to execute commands in remote systems, the "remote radar" command RR of Eros5 and a new "remote esh" command RESH

RR *radarid remotecommand*
RESH *remoteid remotecommand*

Only the latter can be used for direct ESH–to–ESH communication.

Both remote commands are built on top of the COMM SEND command. Both commands first open a socket channel to the remote process, then write to the socket, then wait for a reply from the remote process, and finally return the reply to the caller. If an error happens in the remote command, that error is propagated back to the caller in the standard Tcl manner for error handling. While the remote command is executing in the remote process, the caller blocks. This is the synchronous remote command.

There is also asynchronous version of both commands,

RR *eosid* –async *remotecommand*
RESH *remoteid* –async *remotecommand*,

which just write the command to the socket and then immediately return. The caller of an –async command will not get back any information if something goes wrong, not even a notification, unless notification is issued by the targets system as a side effect of the *remotecommand*. The –async command is a way to tell the remote system that the command's sender basically doesn't care about any feedback. Internally, Eros tries to avoid asynchronous commands because of the loss of error feedback. In some situations, -async commands are used internally to avoid the possibility of a deadlock.

Only the RR and RR –async commands have been available in Eros up to now. With Eros6, support for two new options for doing remote commands was added to RESH. Both new options seek to combine the desirable features of synchronous and asynchronous commands. These features are the non-blocking behaviour (immediate return), and automatic error feedback. We refer to these two optional forms of RESH as *RESH background commands* here. The syntax is

RESH *remoteid* –command *callback remotecommand*
RESH *remoteid remotecommand* &

The former type is a simple wrapper over the COMM SEND –command command, while the latter type is a special case of the former, where a certain special *callback* is used internally.

Both background RESH commands write the *remotecommand* to the outgoing socket and then return immediately to the caller, with a unique identifier (serial number) of the command. The receiving end of the socket channel sees these commands as synchronous and will therefore return the value and error status of the *remotecommand* back to the calling RESH, though not directly to the caller, who might already be doing other things at that time. Rather, the return info is given the ESH event loop, which then executes the piece of code named *callback*.

The important thing is that when the *callback* is executed by the event loop, also the full return info from the *remotecommand* is available to be used in *callback*. The *callback* can do whatever is needed with the information, such as write a notification to the ESH alert window or save the return value to a previously agreed place. The plain background command actually does just that. If there is an error in *remotecommand,* an error message is written to the alert window, and also saved so that it can be queried by ESH, based in the command serial number. If there are no errors, ESH just saves the return value for later retrieval by a separate ESH command.

For some use, the background RESH command is an improvement over the plain RESH. One gets error info and even a return value, but one does not need to wait idle when the remote command is being processed somewhere else. But this is not yet very useful in many cases, typically, when these commands are used within a script. In interactive use, the user might be happy to keep an eye at the ESH notification window while doing other things when the remote command is executing, and will be able to see if an error happens, and can then decide what to do.

In a script, it is often desirable to preserve strictly the order of the commands, so that next one is not done before previous one is ready. This is what a sequence of synchronous RESH commands will automatically accomplish, but then one must be prepared to live with blocking. The ideal way would be if one could avoid blocking, would get automatic error feedback, and would still be able to also ensure ordering of commands. In this WP, we have developed such a solution, which we call "*joinable background commands*". The solution requires multitasking within an ESH. When multitasking is available, the game changes.

With multitasking, only the task that is executing the script blocks when waiting for remote commands; other tasks, including user's interactive commands, can proceed normally. For example, with Eros6 multitasking, it is a simple matter to put three tasks that in Eros5 require three separate processes, into a single ESH process:

- Run an experiment script in one *job* (Eros6 uses the word "job" instead of "task")
- Run some kind of monitoring in another job, perhaps displaying output on a separate ESH window
- Still also have interactive terminal input available

Multitasking is most typically in these days implemented using threads. Threads have been available in Tcl since version 8.5 as an add-on, and became part of the Tcl core configuration with version 8.6. ESH provides access to threads in terms of thread pools, but internally in Eros, they find their use mostly on the server side, the EOS (see Section 4.2 for an example). But threads are not how the ESH jobs are implemented. For jobs, Eros uses coroutines. Coroutines are used to implement co-operative multitasking in Eros, within a single operating system thread. Typically, that thread will be the one and only thread of an ESH process.

3.3 Coroutines

Coroutines only became to Tcl with version 8.6, in December 2012. The concept itself dates from at least from early 1960s[1], but has long been out of fashion. From the most widely used general-purpose languages, only Python (since version 2.5 in the form of generators) and now Tcl provide them.

There are various ways to approach the coroutine concept. Here, we are concerned with multitasking. For that purpose, a useful characterization is that coroutines are procedures that can have multiple return points, called the yield points. One can leave a procedure at a yield point, then later came back to find the procedure precisely in the state one left it, all local variables intact, and then repeat this as many times one needs. When we say "leave the prodecure", we mean that the thread of control of the underlying pro-

[1] Conway, M. 1963. *Design of a separable transition-diagram compiler*. Communications of the ACM 6, 7 (July), 396–408.

cess leaves the coroutine, and moves over to either the Tcl event loop (which in this context is best seen as kind a special internal Tcl procedure), or to another coroutine.

The Tcl command to leave the coroutine and move execution to the event loop is YIELD, and the command to move execution to another coroutine is YIELTO. The way to get the control back to the coroutine is either with YIELDO (from another coroutine), or by calling the coroutine by name.

A coroutine is initially created and started by a the special Tcl command COROUTINE, with syntax

COROUTINE *coroname commandname* ?*parameter*...?

The *commandname* is the name of a standard Tcl procedure. That procedure is the piece of code that initially gets called when the coroutine of name *coroname* starts executing. The COROUTINE command creates a *coroutine environment*, with the user-given name *coroname*.

Basically, the coroutine environment is a special kind of callable function in Tcl. That environment maintains the long-living structure that preserves the state of the procedures that are invoked as a result of nested subroutine calls when the initial command *commandname* is called. All these nested routines are said to be executing in the coroutine environment *coroname*.

There is a special command, INFO COROUTINE, which an executing piece of code can use to find out its own coroutine environment. If that command returns an empty string, the caller knows that it is not running in a coroutine environment, and can act accordingly. "Coroutine" and "coroutine environment" are used interchangeably here, but neither one must be confused with the COROUTINE command.

A given coroutine environment exists in the Tcl interpreter until it is deleted in one of two ways. Either it is deleted explicitly by renaming it to an empty string, or the Tcl system deletes it when the initial command *commandname* returns in one way or another. In Tcl, it is possible to execute a given command at any level of the call stack frame hierarchy. It is possible that when one is deep in the nested hierarchy of calls, to force the coroutine immediately to return from deep down, which causes to coroutine to be deleted. To find out if a coroutine *coroname* still is around, INFO COMMAND *coroname* can be used. If it returns an empty string, the coroutine no more exists.

Between the time of the initial COROUTINE command, and when the coroutine is finally deleted, in this report we will sometimes say that a coroutine is *cosleeping* when it has given up the thread of execution, by having YIELDed to the event loop. If it has not done that, we say the coroutine is running. We say that a cosleeping coroutine is *woken up* when it is called by its name, the *coroname*, and the thread of control moves back to the coroutine.

We warn that the term cosleep is entirely non-standard, but we need a short way to say that the coroutine has YIELDed and is therefore not executing user code. We cannot just use "sleep", because that word refers to thread and process level concept, which we also need. When a coroutine is in cosleep, it may well be that the ESH process where the coroutine lives is not in process sleep, but instead is busily running another coroutine. Or the ESH might be in the event loop, and when being there, may or may not be in process sleep.

One of the two main features that make coroutines very handy and comfortable to use in ESH, is that when a particular job is executing, no other job can be executing literarily at

the same time in that ESH (as long as one is not explicitly using multi-threading). If there are other jobs around in that ESH, one knows for sure that they are cosleeping at that time. Simply because an ESH is single-threaded, there is never any need for an ESH coroutine to protect resources from possible simultaneous use by other coroutines.

There are only very few ways that an executing piece of Tcl code can give up the thread of control in the middle of the code. One of the ways is YIELD (and YIELDTO), another is execution of the Tcl command VWAIT. As long as these command are not present in a piece of code, either directly or via some of the called subroutines, a person looking at the source code can know with certainty that there will be nothing else executing in the system at that time. That is, most coroutine code in Tcl behaves automatically, without any special protection measures, analogously to "critical sections" of thread programming. This makes writing and understanding multitasking code simpler.

The other useful aspect of Tcl coroutines, compared to Tcl threads, is that all the coroutines have direct access to all the same routines and variables as normal Tcl commands in that interpreter have. Each Tcl thread, instead, executes in its own interpreter, which has its own private function and variable space, which initially is very empty. If any manipulation of global variables, say, is needed, coroutines are a good choice.

A potential drawback with coroutines as a general programming tool is that they all live in a single thread, and so cannot literally be executing simultaneously even on a multicore CPU. But this is an issue at most when really high-performance computing is called for. Performance *might* become an issue with E3D occasionally; but even there probably only on the server, or EOS, side of Eros.

On the ESH side, in normal interactive use, or when running experiment scripts, the single OS thread that runs the ESH interpreter is idle (in OS sleep) most of the time. When an ESH is running an experiment script as a coroutine-based job, most of the time the job is cosleeping, that is, has yielded to the event loop. The Tcl event loop is programmed very efficiently; when an ESH is not running user code, the CPU usage will be a of the order of 0.01% in a decent modern CPU.

3.4 ESH multitasking via coroutines

In ESH, coroutines are made available via special wrappers and are called *jobs*. Commands to start and stops and list jobs and show their state, are provided. Jobs can have their own output windows. Experiment scripts are run as jobs. Jobs can GOTO (JMP) from one block in the experiment script to another by an interactive user command.

The way to get many jobs running simultaneously (but keep in mind that they *actually* use CPU time one after each other, in a co-operative way) is to give the initial COROUTINE command via a SPAWN or a RN command, and wait that command to return, and then launch the next job. It must be noted that "*coroutine returning*" and "*COROUTINE returning*" are quite different things. If one means anything with the first phrase, it should refer to the moment when the coroutine environment gets deleted, while the latter means the same thing as with any other Tcl command. There is room for unpleasant surprises here.

To achieve multiprocessing, it is necessary that the COROUTINE command returns before it is deleted so that one can launch a new coroutine before the previous one vanishes. The COROUTINE command returns when the coroutine code hits a YIELD command. The COROUTINE command's return value is the parameter that sits on the right-hand side of that first YIELD command in the code. If nothing is there, an empty string is returned. If there are no YIELD commands, the COROUTINE command returns when the initial command *commandname* returns.

The COROUTINE command returns as soon as, but not before, the newly created coroutine environ for the first time goes to cosleep.

> In ESH, a job only goes to cosleep due to one of four reasons: the three synchronizing commands AT, SLEEP and SYNC, or a synchronous remote command. All ESH commands accessing radar hardware via an EOS are of the last type.

We are undecided what should be done with the standard Tcl VWAIT within a coroutine. Here as elsewhere, the VWAIT command causes the thread of execution go to the event loop, but if one tries—using a remote command from another ESH for example—to call the coroutine by name, one will get a Tcl error message saying that the coroutine is already running. This is consistent with our nomenclature in Section 3.3 that a coroutine is (co-)sleeping when it has YIELDed to the event loop, and is running otherwise.

In ESH, user programs should not explicitly use COROUTINE, VWAIT or YIELD if jobs are also used within the same program (but this is not enforced). That is, either you let ESH manage coroutines for you via the job construct, or you manage the coroutines entirely by yourself. Experiment scripts anyway run as jobs if they are started with the RUNEXPERIMENT command.[2]

To get coroutine-based multitasking in ESH, all participating jobs must go to cosleep every now and then. If a job is started and never goes to cosleep, even the interactive use of ESH is prevented.

The Eros6 command to start a new job, by executing the Tcl COROUTINE command, is the SPAWN command, which has roughly the following syntax

SPAWN ?-j *jobname*? ?-w *winname*? ?-t *starttime*? *commandname* ?*arg*...?"

Only the name of the coroutine's initial command *commandname* and its possible parameters are obligatory, defaults for everything else is provided. By default the job will have its own output window, so that the job output does not mix and mesh with interactive user input. The *commandname* that specifies the initial procedure that the coroutine environments starts to execute, is by default assumed to be an Eros BLOCK, but can also be an ordinary plain PROC. It is recommended that Eros6 jobs use BLOCKs when possible.

In Eros5 scripts, the BLOCKs have been the normal way to define subroutines in experiment files. They have been serving two main purposes there by providing wrapper code both at the beginning and at the end of the standard Tcl PROCs. First, the wrappers allow automatic bookkeeping by Eros, such as keeping automatically track of in which one of the nested BLOCKs the thread of execution is at a given moment. And they have helped in the implementation of the GOTO context switching command. In Eros6, the wrappers are no more needed for the context switching, but they still do useful bookkeeping.

---

[2] For the future multi-user, multi-experiment, possibly highly interactive E3D environment, it is far from clear what should constitute an "*experiment*". It seems possible that *experiment* could become just an accounting concept, based on an allocated time slot and radar resources. During that time slot, it would be up to the user how he generates commands to the radar.

The *starttime* parameter of SPAWN initializes Eros synchronization time counter to the given instant of time, but the SPAWN command itself is executed immediately. Any scheduling of the SPAWN command itself must be done with the (lowercase) AT command. The AT command works much the same in Eros6 as in Eros5. In ESH, it is legal to use AT also in scripts, but such scripts are not downward compatible with Eros5.

The status of a job can be displayed using the Eros JS command.

In the current, version 1.0, implementation of ESH, a job must be finished-off explicitly using STOPJOB command. The underlying coroutine may or may not have returned and been deleted at that time. The reason for this maybe odd implementation is that at some point Eros needs to clear the descriptor table of that job, and the reasonable time to do that is when the stop is stopped. But one of the pieces of information that is present in the job table, and can be printed out with JS, is the job status and the current return value (the return value of last executed command, as is the normal Tcl practice). One might want to see those values after the job has been stopped; they are kind of job exit status, and the job return value. We want to keep those values available, so must not let the job table vanish automatically when the coroutine gets ready. Therefore, Eros only clears the tables after the STOPJOB command has been explicitly given. On the other hand, these are early days in our experimentation with coroutines and multitasking in Eros, and the implementation may change in the future.

We described above how the JS command could be used to retrieve a return value of a job. We emphasize that in general the jobs, implemented via coroutines, cannot be used to return a value in the normal way that Tcl commands return values. As we have said, the job command (if it is suitable to co-operative multitasking in the first place), returns more or less immediately, when it first goes to cosleep. At that time, it may return a value, but the value is normally not any kind of end result of the job, but only the parameter of the first YIELD in the job. Coroutines can be used to good benefit also to return values, but not normally in the present context where they are used to implement multitasking.

The coroutine-based multitasking in Eros6 is co-operate multitasking. Unless all jobs go voluntarily to cosleep every now and then, the first one that does not will get all of the CPU time that the operating system gives to the ESH, and all the other jobs freeze. But the typical structure of experiments is such that there are ample opportunities for a job to go to cosleep. One of them is via the SLEEP command. (Both AT and SYNC are then implemented calling SLEEP.)

Box 1 shows in pseudocode how the Eros6 command SLEEP *milliseconds* is implemented. The actual Tcl implementation is in file esh/lib/eshlib.tcl.

The strategy is to define a callback (point (2) in Box 1) that, when called, will wake us up after we have gone to cosleep; then arrange (point (3)) the callback to be called after the required sleep time; and then immediately go to cosleep by yielding to the event loop (at point (5)).

When something finally wakes us up, we check (point (6)) whether it was the timer (normal case), or if something exceptional happened while we were sleeping. In the normal case, we return to the caller of the SLEEP so that a job's execution can continue, else we handle the exception as required. For the “exceptional” case, see Section 3.6.

We note the somewhat obscure semantics of the YIELD command. In a sense, that command internally “returns twice”. First *out*, to the caller of the COROUTINE (the event loop now), optionally delivering a value there; next *in*, from the event loop back to us, delivering a value to us, too. The coroutine manual page is pretty dense hereabouts.

Box 1. Implementation of Eros6 SLEEP for coroutines.

```
proc SLEEP millisec

    1) Find our coroutine environment name. It would be an error if we
    were not in a coroutine, but this is checked on a higher level already.
      coroname ← info coroutine

    2) Create a callback script that calls ourselves (wakes us after a cosleep).
    The string "timer" in the wakeup call will show up as the yield return
    value in (5).
      callback ← "coroname 'timer'"

    3) Set up a timer event that triggers after millisec and makes the wakeup call.
      timer_event ← after millisec callback

    4) Update the job state bookkeeping.
      jobstate(coroname) ← "sleeping"

    5) Yield to the event loop. When we wake up, yield delivers a value
    that indicates the reason for the wakeup.
      reason_for_wakeup ← yield

    6) Check the condition that woke us up, and act accordingly.
    if reason_for_wakeup = "timer"       Normal wakeup from timed cosleep.
      jobstate(coroname) ← "running"
      return  to caller
    elseif reason_for_wakeup = "stopjob"   An asynchronous STOPJOB has been given.
      cancel timer_event
      jobstate(coroname) ← "stopjob"
      rename coroname ""       Mark the coroutine as deleted. It will no more
                               show up in 'info commands coroname' after this point.
      return to_top_level      Exit the coroutine by returning via top level of call stack.
    else
      Internal error, we should never come here.
      Print out a warning message.
```

In Eros6, the AT and SYNC synchronisation commands are implemented calling SLEEP with a suitable parameter value. In Eros5, prior to Tcl 8.6, when coroutines were not available the AT and SYNC commands were implemented using the Tcl VWAIT command. The SLEEP, if such a routine had existed in Eros5, would have been implemented as shown in Box 2. This could also be used in Eros6 if the functionality is needed in a non-coroutine environment (but the SLEEP command is only allowed within jobs).

This looks simple enough, but leads to subtle, though well known in the Tcl community, difficulties in multitasking. We need to explain this matter now, because one of the main reasons for introducing coroutines into Eros is to avoid the VWAIT command as much as possible.

Box 2. SLEEP as if in Eros5, using vwait

```
proc Tcl8.5_SLEEP millisec
    Initialise a global variable to a catch-all value.
      global reason_for_wakeup
      reason_for_wakeup ← "illegal"
    Create a callback to set the variable.
      callback ← "reason_for_wakeup ← 'timer'"
    Create a timer event to invoke the callback.
      timer_event  ← after millisec callback
    Go to the event loop. Return here when the variable is written to.
      vwait reason_for_wakeup
    Find the reason for the wakeup, and act accordingly.
      if variable = "timer"
        Timer expired, return to caller.
      elseif variable = "stopexp" or variable = "gotoblock"
        The legal exceptions.
        Cancel timer.
        cancel timer_event
        Force an error in a controlled manner
        so that call stack unwinds, and then proceed from there.
      else
        Should not come here.
```

Already in Eros5, there is kind of multitasking going on, (though, in Eros5 only on the server side) achieved by means of the event loop and the VWAIT. If one tasks, call it *Task1*, e.g. a command received from a client, goes to event loop via VWAIT, other command, *Task2*, from some other client, can arrive, and the server can start serving it. It will serve *Task2* until that *Task2* either gets ready (in which case control is can be returned to *Task1* if required and things can proceed normally), or *Task2* in turn issues a VWAIT.

The problem arises with this second VWAIT, issued by *Task2* while the VWAIT given in *Task1* is not yet ready. The problem is that the two VWAIT commands are not independent but nested. The result is that the first VWAIT cannot return to its caller, even if its variable has been written to, before the second VWAIT gets ready. The horror scenario is that the *Task1* issues a VWAIT that should return in, say, two seconds, but during that time, when thread of control is in event loop, *Task2* arrives to server and issues a VWAIT that will expire after ten seconds. Due to the nesting, the result is that *Task1* needs to wait ten extra seconds, until the VWAIT of *Task2* expires, before it can proceed. An example of such nesting is shown in Box 13 of Chapter 4.

This clearly is unfortunate behaviour in a server environment, but there seems to be good reasons why a WVAIT-like functionality must have been implemented as nesting call prior Tcl 8.6. Tcl wiki.tcl.tk/1527 explains the event loop and how the VWAIT is done. Basically, the event loop is a (C-) routine, call it EventLoop, which inside it has an infinite loop. A pass of that infiloop checks if there are events (callbacks to execute) in the event queues, and if there is not, the infiloop sleeps for a while, then tries again. If there are events, they are processed and then the idle looping presumes. The VWAIT sets the EventLoop state so that the only event that actually is handled is the one where the vwait variable is set. When that happens, normal execution inside that version of EventLoop presumes. Cleverly, after setting the EventLoop state like that, VWAIT *makes a recursive call* to the EventLoop routine. This gives a fresh version of EventLoop, but one level down in the call stack. That fresh version allows all events gain be handled, and so normal event handling can continue, but one level down the stack. It is not stated in the

wiki, but it is seems that the good and necessary thing in the recursive call is that the current state of the routine that made the VWAIT call gets automatically saved in the call stack while the thread of control leaves the routine. This is of course necessary, because the new events may call the original, vwait-calling routine, and its local variables must not be overwritten by the new call.

Thus, the ultimate reason for the VWAIT behaviour lies in the recursive nature of the Tcl central engine prior to Tcl 8.6. For Tcl 8.6, the whole Tcl interpreter mechanism has been overhauled, to use what its authors called a "non-recursive engine". This, in a sense, allows recursive calls to be done non-recursively, and has made it practical to implement the coroutine system, among other things.

With coroutines, it is actually possible to re-implement also the VWAIT command itself in a way that is perfectly symmetric between is callers, so that no nesting any more happens. The routine COROUTINE::UTIL VWAIT (itself coded in plain Tcl) in the Tcl standard library does just that. The central piece of the tcllib library routine is roughly as shown in Box 3.

Box 3. Coroutine-based implementation of vwait (from coroutine::util)

```
proc coroutine::util::vwait varname
    ...
    1) Get our coroutine name, coroname
    2) Set write-trace on the variable varname
    3) Set up a callback to wake us by calling coroname when the trace triggers
    4) Yield to the event loop
    5) Delete the write-trace
```

To implement the basic synchronization primitive SLEEP in Eros6, we could have used the UTIL::VWAIT of Box 3 in place of the standard VWAIT used in Box 2—and that is still a fall-back option for us—but we chose not to. For one thing, using variable trace (point (3) in Box 3) seemed an unnecessary complication for the task at hand, and for another, having our own implementation allows more detailed control at and around the key YIELD command. This is useful both for debugging and for run-time bookkeeping in general. And for a third thing, if there are multiple, or more complicated conditions, we have more flexibility in handling them. There is a UTIL::AWAIT command to handle OR-like conditions on multiple VWAIT variables, but we want to have still more flexibility.

But we acknowledge that our implementation of SLEEP in Box 1 is somewhat fragile. Our approach requires that we can keep track precisely on each and every possible case in Eros that can cause a cosleeping coroutine wake up from the cosleep, so that we can handle all those return conditions that can arise when the YIELD in SLEEP returns back to the coroutine. This should be possible in our own ESH code. But will we be able to guarantee that for library routines and for user-code?

Besides the AT, SYNC, and SLEEP, the fourth place in Eros6 where the coroutine goes to cosleep is in our re-implementation of the basic synchronous remote command in Eros, the SEND command. The new, coroutine-based SEND is called CSEND. By use of coroutine, we avoid using VWAIT in Eros6 jobs when they wait for answers to the their remote commands.

CSEND makes use of the asynchronous "COMM SEND -command *callback command*" version of the SEND command, which we mentioned in Section 3.2. Now we again define the callback so that it picks the return status/answer from the remote command, but instead of placing it to a variable for asynchronous access, we actually deliver it to the caller. This requires the caller to (co-)sleep until the answer becomes available, that is, we YIELD to the event loop. We use the return value of the YIELD command to deliver the answer to us, and then to our caller. The actual routines used are CSENDCALLBACK and CSEND in esh/lib/eshlib.tcl. The implementation in pseudocode is given in Box 4.

Box 4. Coroutine implementation of synchronous send CSEND

```
proc CSENDCALLBACK coroname args
    Pick the answer/return status from a standard place in args where the COMM
    has delivered them behind the scenes when it calls this callback.
      ans ←... args
    Wake up the coroutine by calling it by name, using ans as parameter.
      coroname ans

proc CSEND remoteid command
    Get our coroutine name
      coroname ← info coroutine

    Prepare callback that will wake us up and deliver the remote answer to us
      callback ← "cSendCallback coroname"

    Send out the command. Send –command returns immediately.
      send remoteid –command callback command

    Yield to the event loop to wait the remote command to execute.
    When it has the answer ready, COMM calls the callback and we come back here.
    When we come back, the answer is available as the output value of the yield command.

      ans ← yield

    Handle the return

    if ans == "stopjob"
      Handle this ...
    elseif
      Normal case, return answer from the remote call to our caller.
      return ans
    else
      Should not come here.
      Print a warning.
```

Calling CSEND, or some variant like RESH, is the standard way that Eros6 performs synchronous remote commands. This completely avoids the VWAIT problem within jobs. And in Eros6, experiments are always run within jobs.

3.5 Joinable parallel commands in ESH

We described earlier an arrangement of doing remote commands in the background, which allowed running many remote commands in parallel. However, there we offered

no way to do that synchronously with other activities. The remote commands came ready when they did, and they left their return values to variables where the values could be fetched. But that is not yet the ideal situation for parallel programming, because we did not offer any way to know programmatically when those commands actually became ready, so that asking for the return value makes sense. This would require some kind of polling arrangements to fetch the values.

With coroutines, a much better way of synchronisation is possible, and requires no polling. We call our arrangement *joinable background commands*. The typical usage scenario is the following. Assume a job is running a script. Then, within the job, any number of Eros5 commands can be launched so that they start working in parallel, at their own pace, and will get ready in whatever order, which one does not need to know in advantage. Assume there is a critical point in the script such that all those commands must be ready before it makes sense to continue executing further commands. Typically, this would happen in an experiment initialization phase, where we want to initialise the several independent subsystems, and want do that as fast a possible. But all the subsystems must be ready before we can actually collect data.

Borrowing an idea from thread programming, we have implemented the command BGJOIN, which is normally called so that it provides a value, as

ans ← BGJOIN *bghandle*

This command is implemented in such a way that when a group of those command is placed after each other in the script, the scripts blocks on that place until all the background commands referred to in *bghandle* parameters, have become ready, and their return values have been collected to the *ans* parameters. To be able to use BGJOIN on the job, that job must have been launched using a special command, BG, with the option *joinable*, as

bghandle ← BG –joinable *commandname*.

The BG command basically creates a new, unique, use-only-one coroutine, with *commandname* as the coroutine command, and a serial-number-based internal name for the coroutine name itself. The *bghandle* allows access to the return value of the command *commandname*.

This may not appear quite as good as running the commands in parallel using threads, for some co-operation from the part of the commands is required. The commands that are placed in background must go to sleep fairly soon. But all commands that needs something from EOS belong to that class, for they normally send their requests to EOS using the new CSEND version (Box 4) of the synchronous SEND, and so YIELD to the event loop sooner or later. If a command does not need to ask anything from EOS, it probably is so simple that there is no point to put that into the background in the first place. One must also keep in mind that the coroutines do not run literally in parallel, they all use the same single ESH thread. So putting commands into the BG mode helps speed-wise only if the commands go out somewhere where they actually are able to proceed independently, really in parallel. This typically happens in the experiment initialization phase.

Moreover, using threads in this context is not really a good option even if one would like it for speed reasons, at least not without extensive re-coding of existing Eros commands. The threads do not know about the variables and procedures in ESH that the commands might need, for each thread runs in its own interpreter. Coroutines instead can run in the same interpreter, and all of them see the same set of global variables and functions.

3.6 Context switching in ESH

As might have been noticed from the code examples of Section 3.4, all the basic commands that handle the wake-up of a coroutine from cosleep, handle the special return value of YIELD marked "stopjob". In Eros6, triggering that condition is the way to perform asynchronous context switch, either the asynchronous GOTOBLOCK command or the STOPEXPERIMENT command.

As we have explained in [DSR2009], by a context switch in Eros5 we mean breaking out from one BLOCK and moving to another BLOCK in an experiment script. The means to achieve that was the GOTOBLOCK command. In Eros, a synchronous GOTOBLOCK is one that is coded in the script. The synchronous jump takes place once the thread of execution comes to that command in the code. The destination is another BLOCK, so a synchronous GOTOBLOCK is not much different from a subroutine call, expect in the sense that it never returns to the caller automatically. If the jumped-to BLOCK returns, the actual thread of execution returns to the event loop, but Eros5 pretended that the thread of control still was in the jumped-to BLOCK, just doing nothing there.

In Eros6, the GOTO command moves the thread of control between coroutines. This is downward compatible in terms experiment scripts, for in Eros6, experiments are automatically run as coroutines. The synchronous GOTO is implemented more or less directly using the new YIELDTO command, which is almost custom-made for our purpose, to jump from one coroutine to the next. Thus, synchronous GOTO is straightforward implement in Eros6, and we will say no more about it here.

Asynchronous, or external, or interactive, GOTO is still a challenge also in Eros6. This is, of course, lamentable because it is of course the asynchronous GOTO that is needed in order to response to external conditions when they occur. Note that though we use the same command name for both the synchronous GOTO and the asynchronous GOTO, conceptually, and programming-wise, they are very different things, and would really merit a different name also (tentatively, we are using "JUMP" in Eros6 for the asynchronous version, though the GOTO can also be used for compatibility).

The challenge in implementing an asynchronous GOTO is how to divert the thread of control out from the script by giving an external command. As we have emphasised, the ESH has only a single thread of execution, and when it is executing some user code, precisely nothing external can happen (under user control); ESH is completely non-responsive during that time to user actions. The way computing systems normally handle this kind asynchronous demand is provide some means for interrupts. These always require some deep-down support from the operating system. The OS *can* interfere with do doings of any thread if it so desires. But the standard Tcl does not provide such a support. There does exist a Tcl extension for Unix systems, called TclX, which provides an interface to interrupts, and in fact we make use of it to disable control_C in ESH.

We have not made any thorough study on this, and a solution might be found in the future, but at the moment is does not look that even TclX could help in providing the control we require for the jump. Even if it might possible to interrupt code in C-level, on Tcl interpreter level, without some kind of explicit support from the Tcl core, one cannot count on just breaking a Tcl command maybe somewhere in the middle, and expect that no bad things follow from it.

So, at last for this time, for asynchronous GOTO, Eros6, like Eros5, still require cooperation from the running scripts. The key to the solution is that in practice, most if not all experiments spend most of their time cosleeping in the event loop. They go to the sleep as a consequence of AT, SYNC, SLEEP, or a synchronous remote command. When

the thread of execution of the script is in the event loop, asynchronous events might be generated by other event sources and reacted upon.

In Eros5, the GOTO command sets the VWAIT variable in AT and SYNC that had resulted in branching to the event loop, to a certain value to signify exceptional event. This then immediately woke the user routine, which then continued execution after the VWAIT. The routine could then inspect the VWAIT variable, and act as needed.

In Eros6, we have many more interrupt points available, namely, all the remote commands also. And as we will see, all the ESH commands that access the radar hardware actually are such remote commands, from ESH to the EOS. As we do not use VWAIT any more in jobs, we take even more straightforward approach to the GOTO in Eros6 than in Eros5. The GOTO command gives the STOPJOB –restart command. This command calls the coroutine by name, with the special value "stojob!" as the call parameter. This wakes up the coroutine at the point where it last YIELDed, and the output value of the YIELD is "stopjob" (see e.g. Box 1). In this way the job finds out that stopping is requested, and can stop itself, returning control to the caller of the STOPJOB –restart. If the caller is STOPEXP, the whole experiment is then finished off. If the caller is GOTO, then the stopped job is restarted (a new coroutine is started, but using the name of the old one), but now using the given BLOCK as to coroutine command. To the user, such stop-start sequence appears as a direct jump from current BLOCK to the requested one.

A small complication is that in multitasking environment, while it is clear *whereto* one wants to goto, one also needs specify, directly or indirectly, which job should do the jump (that is, which job Eros should stop). The default is the "main job"—the one started when an experiment calls its main block—if such a job exists among the current jobs. If there is only one job, then that is of course the one that does the jump. If there are multiple jobs, and none is called "main", then it is an error if source job is not specified. Finally, if there are no jobs active, a new job is started, using SPAWN.

3.7 ESH with multiple interpreters

Even though prior Eros6, Eros has not supported them, Tcl itself has supported multiple interpreters since time immemorial. The concept of multiple interpreters was introduced into Tcl originally for safety reasons in networked environment. The idea was to be able, robustly and strongly, restrict what a remote command, coming over network to our system, could do in our system. Each interpreter works in its own function and variable space, so those can be tailored as needed. Initially, when a WISH is launched, there is only (at least as far as the user is concerned), only a single interpreter, which is called the main interpreter. The interpreter can then create new interpreters, called the slaves, using the [INTERP CREATE] command. A newly created wish interpreter is just the raw WISH interpreter, with nothing extra added to it. So whatever extra is needed, must be explicitly added, in the same way as new commands are added to the main interpreter also when ESH is booted up.

All the interpreters share a single thread, and the single event loop (we think), so multiple interpreters are not a new way to do multitasking. In particular, a single egoistic interpreter can freeze all the interpreters if it never goes to the event loop. Rather, multiple interpreters are a way to keep variable spaces and function spaces cleanly separated. A possible application in Eros6 would be to run each experiment in a freshly created interpreter, then delete the interpreter after the experiment. In that way, the experiment gets a clean initial function and variable environment, and can freely modify the environment without worrying about side effects. On the other hand, it might be simpler to run each experiment in its own ESH instead.

The Eros6 the NEWINTERP command creates a new interpreter under the ESH main interpreter, and initializes it to have more or less same facilities as the main interpreter. Especially, the COMM package is loaded in. Then the ESH main interpreter can communicate with the slave pretty much after a separate ESH, but with the convenience (or nuisance) having it running in the same terminal window.

4 The server side—EOS

4.1 Long-living commands on the server-side

In many ways, the ENGINE (Figures 1 and 2) process is the most critical of EOS processes. Its task is to receive commands that are sent to the radar from ESHes all over the net. Some incoming commands it may send as–is to radar subsystems, by making a remote call to the subsystem, others are composite commands that are basically executed in the ENGINE but may also access one or more of the subsystems several times. Most of the remote commands the ENGINE sends out will be of the synchronous type (its client is expecting an answer).

In more detail, a synchronous remote command, REMOTECOMMAND, that an ESH wants get executed in ENGINE, is implemented like the following. The ESH uses some form of synchronous send (from inside a coroutine, CSEND, else, the plain SEND that still needs to use the problematic VWAIT for waiting), by issuing a command like

SEND *engineid* COMREC *REMOTECOMMAND*

(In terms of Figure 2 about the Eros6 EOS architecture, the communication receiver command COMREC would conceptually be inside the Communications Manager block. Here we, for simplicity, still assume the Eros5 structure, were COMRES is executed directly in the ENGINE process.)

The SEND command causes a chain of events. We are mostly interested in the server side now, but will start from the beginning.

1. The SEND command is picket up by ESH interpreter. The interpreter invokes COMM to handle it. COMM gets two parameters, remote ID *engineid*, and a text string M = "ComRec *REMOTECOMMAND*".

2. COMM of ESH opens a socket channel to ENGINE if one is not already open. Then it writes M to the socket channel. (M goes out wrapped inside the wire protocol package, but we skip over that phase here.) Then COMM goes to event loop to wait for a reply from the engine The wait is done differently in SEND and CSEND, but here we are interested mainly on what happens on the server side, so the mode of wait does not matter.

3. The COMM of the ENGINE gets the string M from a socket channel. COMM may optionally make some initial checks and perform some transformations to the message (which we ignore here), and then evaluates M in the ENGINE main interpreter.

4. The evaluation causes the Eros command COMREC to be called, with *REMOTECOMMAND* as the command argument.

5. The COMREC command (in eros/lib/exprun.tcl) looks as shown in Box 5.

Box 5. Eros ComRec routine.

```
proc ComRec remotecommand
        Do some bookkeeping based on the command name, then
        evaluate synchronously the script in the ENGINE interpreter.
          reply ← remotecommand        (*)
        Return the reply to our caller, COMM.
          return reply
```

Thus, ComRes issues the REMOTECOMMAND and returns the answer (and an error status) to COMM when it gets the answer.

It depends on the REMOTECOMMAND whether or not the wait for the *reply* in (*) also includes one or more periods of sleep in the event loop. If the REMOTECOMMAND script itself issues commands to the radar subsystems using synchronous send, as in Box 6.,

Box 6. A command that mixes remote commands and local calculations.

```
remotecommand = {
                    ans1 ← Send subsys1 Command1
                    ans2 ← Send subsys2 Command2
                    Use ans1 and ans2 to construct a final reply, then return.
              return reply
    }
```

then the SENDs will wait in the event loop, and thus ComRes, which is synchronously waiting sleeps with them.

7. When the REMOTECOMMAND in (5) gets ready and returns, the reply is returned back to the caller of ComRes, COMM in this case. The COMM then writes it to the waiting ESH client over the same bidirectional socket channel where the command came from.

8. The COMM in ESH gets the *Reply*, activates the waiting Send from sleep and delivers the *Reply* to it. The Send becomes ready and returns with the *Reply*.

Our problem on the server side is the following: What should ENGINE do if the REMOTECOMMAND in (*) takes a long time to execute, that is, to return. How are possible other clients served during that time? There are two level of seriousness to this problem, depending on whether the long-living REMOTECOMMAND ever goes to sleep or not. If it never sleeps, we need threads. If it does, we can get good solution using coroutines.

4.2 Serving long-living commands via threads and asynchronous return

If the REMOTECOMMAND in (*) in Box 5 never goes to the event loop, it will monopolize the ENGINE for the whole of its duration, for, remember, the ENGINE is a single-thread system. That means that no other clients can be served during the duration of that long-lived REMOTECOMMAND. This is bad enough in Eros5, but could really become a nuisance in Eros7, where several simultaneously running experiments from several users from a multitude of ESHes all want to be served by that ENGINE.

If the long-lived REMOTECOMMAND does not go to sleep, the only way to avoid freezing the whole ENGINE for all other clients, is to introduce more threads. The idea is to let such a remote command execute in its own thread, instead of executing in the same thread as the ENGINE main interpreter. Tcl 8.6 THREAD package provides what is called a thread pool (TPOOL) for this kind of purpose. Once a TPOOL is created, one can use commands like

TPOOL::POST *REMOTECOMMAND*

to execute REMOTECOMMAND in one of the worker threads which the pool maintains. The TPOOL::POST commands returns immediately, with a handle that identifies the post, so that the REMOTECOMMAND return value and error status can be queried asynchronously, *when they are ready.*

The thread pool idea leaves two problems to be solved. First, we have the usual problem of Tcl threads that it each thread executes commands in its own private interpreter, not the ENGINE's main interpreter. Therefore, each thread has its own function and variable space. If the REMOTECOMMAND would need access, say, to the radar state variables maintained the ENGINE main interpreter, access to it would need to be specially arranged, for example by giving all needed data as call parameters. That is maybe OK for new command, but would be a pain for old, well-established Eros5 commands. They would need to be rewritten.

The second, and more tricky, problem for the ENGINE is that TPOOL::POST only provides the reply to the REMOTECOMMAND asynchronously. Thus, in the command [*] in Box 5, one cannot write something like

reply ← TPOOL::POST *REMOTECOMMAND*

to capture the answer from the REMOTECOMMAND. The *reply* would just be the handle to the post itself. The manual page of the TPOOL says that one needs first to use

TPOOL::WAIT handleofpost

to wait for the posted command become ready, and only then can one use the handle to recover the sought-for reply via the TPOOL::GET command. This mechanism looks much like the BGWAIT we use in the ESH background commands to join the background command.

The man page also says that the TPOOL::WAIT "waits in the event loop" until the requested command is ready. So we would code the COMREC in this case like shown in Box 7.

Box 7. How to get return value of a command posted to tpool thread.

```
...
handle ← tpool::post remotecommand
tpool::wait handle
ans ← tpool::get handle
return ans
```

At first thought, this may seem to solve the problem of the never-sleeping REMOTECOMMAND. The command is now in its own thread and can freely use all the time its thread will get from the OS, without disturbing anybody else. Meanwhile, the ENGINE main interpreter went to the event loop, and is thus ready to serve other clients.

But there actually is still a problem. The TPOOL manual page does not say *how* the TPOOL::WAIT goes to the event loop. As there are no coroutines anywhere to be seen here, one must strongly suspect that the TPOOL::WAIT is done using VWAIT, with all the problems that might ensue.

To test if the TPOOL::WAIT indeed behaves as VWAIT, we implemented (5b) via a new EOS command, SIMPLETPOOL. Then we used two ESH processes as clients for a simulated UHF ENGINE. As the REMOTECOMMAND in Box 7 we used the Eros6 test command FREEZE *N* which fully monopolizes the thread which executes it for that many seconds using Tcl's AFTER *milliseconds* command (which places the thread to process sleep for the specified time).

In the test, ESH(user1) asks the ENGINE to serve the long-lived "FREEZE 2seconds" command using the thread pool mechanism. Then, one second in the command, ESH(user2) asks ENGINE to serve the even longer-living command "FREEZE 10seconds". This is the standard arrangement to provoke VWAIT nesting. The first, shorter WVAIT is masked by a second, longer WVAIT, so that the shorter wait, even while its timer has triggered, can actually only return to its caller when the longer VWAIT gets ready.

Result of the test, copied and pasted from the ESH terminals, is shown in Box 8. For precise control of the required command interleaving, the test commands were not typed manually, but were launched with the Eros6 scheduler command AT, with time parameter *fm*, meaning next full minute. The AT scheduling is accurate within about a millisecond. Only the fractional seconds of the *hh:mm::ss.uuuuuu* timestamps are shown here. The timing is done using Tcl's CLOCK MICROSECONDS command wrapped inside the Eros TIC and TOC commands, which implement the Eros interval timer. The server-side timestamps show explicitly that the TPOOL::WAIT nests just as VWAIT.

Box 8. Test to show that TPOOL suffers from vwait nesting.

As seen by the users

```
esh(user1): at fm {tic; resh uhf SimpleTpool freeze 2; toc -v}
00.001491 11.093036 ==> 11.091545       KEY: Begin Ready ==> Duration
esh(user2): at fm+1 {tic; resh uhf SimpleTpool freeze 10; toc -v}
01.001069 11.092796 ==> 10.091727
```

As seen by the UHF ENGINE (the timestamps come from both sides of tpool::wait)

```
00.083670 SimpleTpool freeze 2: going to tpool::wait
01.083169 SimpleTpool freeze 10: going to tpool::wait
11.091587 SimpleTpool freeze 10: tpool::wait done
11.092064 SimpleTpool freeze 2: tpool::wait done
```

Therefore, while surely an improvement over a complete freeze of service, the simplest use of TPOOL, as in Box 7, is susceptible to the VWAIT malady, and therefore is not an acceptable solution for the EOS ENGINE.

When the TPOOL is used on the client side, such as interactively from the ESH command line, the TPOOL::VWAIT, TPOOL::GET combination is perhaps the best one can do. But in a COMM server environment it is possible to use a clever feature of the COMM package to achieve a much better solution. The feature is called *asynchronous return* in COMM documentation. We have coupled the feature with TPOOL into a new Eros6 EOS command

RTNASYNCTPOOL *REMOTECOMMAND*.

When RTNASYNCTPOOL is used in place of SIMPLETPOOL in the test shown in Box 8, the service timing becomes as shown in Box 9.

Box 9. Test to show that RTNASYNTPOOLsolves the vwait nesting for TPOOLs

Timing on the client side

```
esh(user1): at fm {tic; resh uhf RtnAsyncTpool freeze 2; toc -v}
00.000576  02.008559  ==>  2.007983
esh(user2): at fm+1 {tic; resh uhf RtnAsyncTpool freeze 10; toc -v}
01.000920  11.008582  ==>  10.007662
```

Timing on the server side, printed out by the tpool worker threads when they receive their respective tasks and when the tasks get ready

```
00.003299 Tservice(#9): freeze 2
01.004140 Tservice(#11): freeze 10
02.007130 Tservice(#9) done.
11.006938 Tservice(#11) done.
```

There is no more any VWAIT nesting. The two worker threads proceed independently and are able to return their answers to their clients without delay, that is, after 2 seconds and 10 seconds after they have been issued, irrespective *when* they are issued. This is the correct server behaviour.

To pull out this happy result, the COMM RETURN_ASYNC command is used. This command creates a command, called the *future object*, or just *future*, and returns a handle to it. Execution of the RETURN_ASYNC command achieves two things. First, it tells the COMM channel that would receive the reply from the REMOTECOMMAND in Box 5[*], to ignore *that* reply. And, second, COMM packs the *future* with enough information about the pending channel that the *future* will be able, when explicitly asked later, feed the sought-for reply to the socket channel in the correct format, *as if* it would be the reply directly from [*]. If the channel has vanishes when the future tries to write to it, special action can be taken. This all happens on the server side; the client of the socket channel will never know anything peculiar having happened when the answer finally arrives to it.

Our implementation of the asynchronous return from remote commands executed in worker threads in a TPOOL is in terms of two routines, the routine RTNASYNCTPOOL that the clients can call remotely, and which lives in the ENGINE main interpreter, and a routine TSERVICE, which is known only to the worker threads. The implementation, in six lines of code, is shown in Box 10a–b.

Box 10a. Implementation of EOS server with TPOOL and COMM return_async

proc TSERVICE Mainthread Future remotecommand

This routine, Tservice, executes in one of the worker threads of the thread pool. Remember the restriction that, therefore, by default we do not have access to the ENGINE's variables or functions. So remotecommand cannot refer to them. Keeping this in mind, execute the remotecommand, getting a result.

result ← **remotecommand**

Now we need initiate action to get the result delivered to the client socket that is expecting it. In the server's main interpreter, there now lives the object Future, which can do this service. We have its handle, Future here; it is actually a command name. The trick is to execute the command Future in the server's interpreter, with the return subcommand and the remotecommand's result as the parameters. Threads in Tcl in general, not only in tpool, can send commands to other threads in the same wish process for execution. We do it now, and to be safe, do it asynchronously so that no deadlocks can result in the message passing. We define the required command string, with the name DeliverTheResult,

DeliverTheResult ← "Future return result"

and send the string to the server's main interpreter, which runs in the Mainthread

thread::send *–async Mainthread DeliverTheResult*

We are done here, and the worker thread returns to the pool to wait for next task.

Box 10b. (continued from Box 10a).

proc RTNASYNCTPOOL remotecommand

Create the tpool if not already done, then:
Ask COMM *to prepare for async result delivery, relevant for presently pending socket in* COMM*.*

Future ← **comm::comm return_async**

Get our thread id so that the Tservice in tpool will be able to send a command to our (= the server's main) thread

me ← **my_tread_id**

Post the remotecommand to the tpool together with delivery info. The tpool::post command returns immediately.

tpool::post Tservice me Future remotecommand

The ENGINE has now set up the necessary framework both for the execution of the remotecommand, and its asynchronous delivery to the appropriate client. The ENGINE is from now on free to serve other clients.

4.3 Serving long-living commands via coroutines and asynchronous return

As we have noted earlier, if coroutines can be used instead of threads, the problem of data and function sharing is simplified. For the EOS ENGINE the limitation that the threads to not have an easy access to the radar state is serious. This is so mostly for the practical reason that in Eros5 has a large set of commands that one might execute in the server, and they sometimes do and sometimes do not need access to the global state. Eros6 needs to make uses of these routines. It would be a non-trivial effort to ensure that they all work in the by-default private variable and function space of the threads. It would be better if the old, tested routines could be used completely as they are. Most, (actually, we think, all) of the relevant routines in Eros5 are either short-lived, or if long lived, do not need much processing power from the ENGINE. They rather make remote calls to subsystem, and for most of their time in, ENGINE they are "sleeping". As we have seen on the client side, this kind of behaviour is well suited for co-operative multitasking via the coroutine mechanism.

In Eros5 ENGINE, commands can be sleeping in the event loop as a consequence of a VWAIT invoked by a synchronous SEND, and so there is the danger of VWAIT nesting. But it is a simple matter to convert the Eros5 SEND to Eros6 CSEND, by testing immediately after entry to SEND if the SEND is executing in a coroutine environment, and if it is, just call CSEND. Thus a change is need only in a single place in the server code.

But to use the CSEND, we need coroutine environment. Thus the problem is how to arrange the *remotecommand* that arrives from a client to the server for execution as in [*] in Box 5, to be executed in a coroutine environment. And this must be done in a way that we get the reply from *remotecommand* back to the client.

Before giving the correct solution, let's consider some of the bad ones that we tried. The first idea was to replace [*] of Box 5 by a coroutine call like

reply ← COROUTINE *coroname remotecommand* [**]

This ensures that the *remotecommand* executes in a coroutine environment.

But did not work. As we have pointed out, if a coroutine yields, it will not return the normal Tcl return value which is the return value of the last command in the called function . The return value of the COROUTINE command in [**] is not the return value of the *remotecommand*, but the value, often just an empty string, from the right-hand side of the first YIELD that occurs when the *remotecommand* starts executing.

So we needed to attempt something more complicated. Our next idea was to make ComRec itself to a coroutine already at EROS boot-up time, then just YIELD out if it then. This leaves the ComRec-coroutine waiting to be woken up when needed by COMM, via a call-by-name. In this way, the *remotecommand* would be called inside a coroutine, but we would not need to use the suspicious COROUTINE command. Sounded very promising.

But did not work either. We never got back the reply we expected from *remotecommand*. It took some time to realize that a stronger version of our rule of thumb, that the COROUTINE command cannot be used to return value, applies here.

The stronger rule of thumb is this: *which ever way* a coroutine is called, don't expect that call to return the value you would like. Calling the coroutine by name again only returned the value from the right-hand side of the first YIELD that occurred after the call. This likely is what the manual page says, too.

Whatever we tried, we could not pull a return value out of a coroutine in the normal synchronous way that functions return a value, as is needed in [*] in Box 5. Was there

any options left? It is remarkable that the COMM package, designed many years before coroutines were anywhere to be seen in the Tcl world, manages to provide just the perfect device for this need, the asynchronous return.

The coroutine-based solution for serving long-lived commands is in terms of two functions, RTNASYNC and RTNASYNCWORKER, and uses the same structure as the thread pool based solution shown in Box 10. The coroutine-based solution is explained in Box 11a-b.

Box 11a. Implementation of EOS server with COROUTINE and COMM return_async

```
proc RtnAsyncWorker Future Remotecommand

    Execute the remote command on the server's main interpreter. One has all functions and
    globals available, so any command that is at  all known to the server works here as is, but
    is now running in a coroutine environment.
            reply ← Remotecommand

    Call the future to return the reply, asynchronously, to the socket leading to the client.
            Future return reply

    Exit the coroutine, by returning via the top level of call stack
            return –level [info level]
```

Box 11b. Continued from Box 11a.

```
proc RTNASYNC  Remotecommand

    Each served remotecommand will need its own unique coroutine environment. In order
    to ensure that when the routines are woken from sleep by calling them by name, there
    will not be any confusion whom to call. We append a serial number, n, to name template.
            global n
            n ← n + 1

    Ask COMM to prepare the framework for delivering a return value, some later time, to the
    currently pending socket channel.  The code after receiving the remotecommand in COMM,
    up to this point, has had the tread of control all for itself, so there can be certainty about
    which socket channel one is dealing with here.
            Future ← comm::comm return_async

    Create a new coroutine environ, with a unique name, and use that environ to execute the
    RtnAsyncWorker. (The RtnAsyncWorker will execute the remotecommand, and then call
    the future to deliver the answer to the socket client.)
            coroutine RtnAsync_n RtnAsyncWorker Future Remotecommand

    The coroutine command returns as soon as remotecommand yields for the first time.
    Whatever value it returns here, will not be used, because the return async command
    above has just told COMM to ignore it.  We don't even look at it, and just formally return
    nothing to COMM here.
            return
```

Maximum gain from the coroutine based asynchronous return is achieved when the clients that simultaneously request service issue remote commands that can execute truly in parallel. A typical case might be that the remote commands are sent to two independent radar subsystems.

We can inspect this situation by having four ESH processes, which we call user1 and user2, and sbsys1 (remote id 5001) and sbsys2 (remote id 5002). As the remote command to be given to the subsystems we use the "FREEZE n_seconds" command. The EOS is a simulated UHF system. The time-stamped output on the four ESH terminals is shown below as copied here from the ESH terminal windows. We have simplified a little the ESH prompt strings, and removed the *hh:mm:* part of the *hh:mm:ss.uuuuuu* timestamps which corresponds to the "next full minute", *fm* given to the Eros scheduler AT command. Console output of the four involved ESH processes is shown in Box 12.

Box 12. Testing coroutine-based EOS server implementation (command RTNASYNC)

Client view. Two esh *processes issue long-living remote commands to two "subsystems" which in this simulation are also esh processes, listeting on ports 5001 and 5002.*

```
esh(user1): at fm {tic; set ret [resh uhf RtnAsync resh 5001 freeze 2]; toc -v}
00.001570   02.012289   ==>   2.010719

esh(user2): at fm+1s {tic; set ret [resh uhf RtnAsync resh 5002 freeze 10]; toc -v}
01.001554   11.011923   ==>   10.010369
```

Subsystem view. Console output of the two esh'es while the freeze command is executed.

```
00.005650 entry sbsys1, freezing for 2 seconds
02.008641 exit sbsys1

01.005653 entry sbsys2, freezing for 10 seconds
11.008743 exit sbsys2
```

Clearly, both users got serviced pretty much like the other one would not have existed. The typical VWAIT-unfriendly timing for the user commands was used, where the 10 seconds command was issued sometime after the 2 sec command, but before the shorter-lived had returned.

If we leave the RTNASYNC out of the chain of commands, the server will not use the coroutine wrappers, but executes the "RESH *id* FREEZE *secs*" commands directly. In that case, RESH is reduced using the basic synchronous COMM SEND, which uses VWAIT to wait for the remote reply. The result, shown in Box 13, is as expected, the VWAITs nest and the shorter command given first cannot return before the longer command also gets ready. This is the Eros5 server behaviour.

Finally, for comparison, we also used the TPOOL approach in EOS to propagate the remote commands to the subsystems. This test is shown in Box 14.

In these tests, both the thread pool and coroutine-based way of serving the long-lived remote commands worked equally well, and there is no performance difference within

the error bars of the crude timing used here. Both ways are new in Eros6, and finally solve correctly the problem of handling long-living commands by Eros servers. To be able to get rid of the, in this case, up-to 10 seconds of unnecessary waiting that the user1 in Box 13 needs to suffer on top of the 2 seconds that his command actual requires, is perhaps the most interesting result of the WP12. The thread-based way could have been done in Eros5, and is more or less standard Tcl textbook stuff. Our coroutines-based implementation of the EOS server we have not found in the standard Tcl sources yet.

Box 13. Eros5 server behaviour when two clients make simultaneous service requests.

As seen by users. User1 suffers from vwait nesting on the ENGINE *server. His two-second command takes 11 seconds to return.*

```
esh(user1): at fm {tic; set ret [resh uhf resh 5001 freeze 2]; toc -v}
08:54:00.001328  08:54:11.010235  ==>  11.008907

esh(user2): at fm+1s {tic; set ret [resh uhf resh 5002 freeze 10]; toc -v}
08:54:01.001163  08:54:11.010250  ==>  10.009087
```

As seen by the subsystems. Both subsystems become ready normally, in two and 10 seconds.

```
08:54:00.004905 entry sbsys1, freezing for 2 seconds
08:54:02.007532 exit sbsys1

08:54:01.004622 entry sbsys2, freezing for 10 seconds
08:54:11.007596 exit sbsys2
```

Box 14. Two clients served via threads (for comparison to coroutines, in box 12).

As seen by clients

```
esh(user1): at fm {tic; set ret [resh uhf RtnAsyncTpool resh 5001 freeze 2]; toc -v}
00.001684  02.011790  ==>  2.010106

esh(user2): at fm+1s {tic; set ret [resh uhf RtnAsyncTpool resh 5002 freeze 10]; toc -v}
01.001415  11.095903  ==>  10.094488
```

As seen by the EOS server

```
00.004974 Tservice(#21): resh 5001 freeze 2
01.082245 Tservice(#23): resh 5002 freeze 10
02.010164 Tservice(#21) done.
11.093857 Tservice(#23) done.
```

As seen by the subsystems

```
00.005877 entry sbsys1, freezing for 2 seconds
02.009198 exit sbsys1

01.089477 entry sbsys2, freezing for 10 seconds
11.092518 exit sbsys2
```

5 Array manager simulator and ESH pool

We argued in [DSR2009] that the E3D system will not be qualitatively very different from the current EISCAT systems. One obvious novel piece in Eros for E3D would be a module to manage antenna groups. One of the currently floated hardware configuration options is to have 109 antenna groups at the core site. From Eros point of view, it seems clear that some kind of Array Manager module (AM) is called for (one of the "Subsystem managers" in Figure 2). An array manager in turn would have under it 109 antenna group managers (AGM). The antenna group managers would keep status of their group of N elementary antennas, uploading the software needed for beam forming, and so on.

The (cancelled) Task 12.1 of the WP12 was to make sure that having such amount of new communication nodes does not cause any unexpected troubles for Eros. The task was removed from the task list because, for one thing, to conclusively test these kind of issues, would require access to a fair number of nodes on one hand. And for another hand, from our experience with Eros5, we were confident that a few hundred new nodes would not be a qualitatively new step for Eros, which presently handles about 10 nodes at ESR.

There is a need to start building some kind of E3D module into the Eros simulator. In this work package, we have developed a quite powerful and easy-to-talk-with communication node, the ESH process. It seemed sensible to use ESH as a building block for an AGM simulator. We wanted to put 109 ESHes running in a mid-range Mac mini workstation to play the role of the 109 antenna group managers. The workstation was in a different building than the high-end Mac mini that was the host for the rest of the test system.

This many ESHes one cannot any more easily be managed manually, so we added a few commands to ESH, to manage an "ESH POOL" (EPOOL). One ESH, the *EPOOL host*, forks out the desired number of pool members using "EXEC ESH *options* &". The pool members start listening at some port number each. In the present initial version of EPOOL, the port numbers are assigned by the pool host when it creates the pool members, using the *–p portnumber* startup options of ESH. It would be better to let the host OS to assign the ports. One way to achieve this is to add a startup option to ESH, to instruct the newly created ESH to register itself at a given net address. This would allow the epool host to know the port numbers, and those can then be used to communicate with the pool members from anywhere in the LAN using RESH.

The ESHes belonging to the EPOOL are full-fledged ESHes, except that they do not try to handle interactive input from their controlling terminal, which is the EPOOL host terminal. Like any ESH, each pool member has an initially invisible output window at its disposal, which it can pop up for showing notifications or other output. The epool members are given remote commands via RESH.

It takes some time to create an EPOOL. With the 109 member EPOOL, it took about 45 seconds on a mid-range mac mini before all new pool members were listening. The pool also consumed about 700~MB of memory. A quiet pool does not consume a perceptible amount of CPU power, though (*top* shows all pool members as sleeping).

In addition to the EPOOL to represent the AGM nodes, a simulated UHF system played the part of the rest of the E3D in the test. This other part run on a high-end mac mini, connected to the pool host over the run-of-the mill LAN at SGO. In this test, the antenna manager module was just a new command, AM, added to the simulated UHF ENGINE.

The *getstatus* subcommand of the AM was used as the test command. When serving the *getstatus* command, the AM, in a for loop, send a RESH command to each of the 109 net

addresses where the AGM's are listening, to read back a 150 byte status string from each node. AM collects the per-group status messages to a 16 kB Tcl dictionary string, and delivers it to the "UHF user" who requested it. In the Eros6 spirit, the user operates not the Eros5 UHF console, but an ESH.

We timed the AGM pool access, in six situations, depending on the service mode (Eros5 type, using coroutines, or using threads), and numbers of simultaneous users (one or two). The results are shown in Boxes 15 and 16.

Box 15. A single user accesses the "whole array" via the Array Manager

(a) Eros5-type service by executing "AM getstatus" on the server

```
esh(user1): tic; set D [resh uhf AM getstatus $all]; toc -v
13:57:56.641909  13:57:56.832474  ==>  0.190565
```

(b) Coroutine-based service via "RtnAsync AM getstatus"

```
esh(user1): tic; set D [resh uhf RtnAsync AM getstatus [list $all]]; toc -v
13:54:03.947057  13:54:04.141361  ==>  0.194304
```

(c) Threadpool-based service via "RtnAsyncTpool AM getstatus"

```
esh(user1): tic; set D [resh uhf RtnAsyncTpool AM getstatus [list $all]]; toc -v
13:56:25.509097  13:56:25.711684  ==> 0.202587
```

The most important observation is that nothing very bad happened. No smoke came out of the Minis, SGO LAN did not break down, and Eros did not freeze. Though there *was* occasionally some odd behaviour in the Mac OS when we used threads in large number of trials: we got complaints about too many open files. This never happened with coroutines and the plain calls. We actually checked that the returned status in the dictionaries D was as expected also. The test configuration may be very non-representative, but Eros clearly could handle the 109 separate communication nodes on the LAN without breaking down.

The access times were longer than one would have hoped. For a single user to read the "whole array" took about 0.2 seconds, even though only about 109x150 = 16 kBytes in total was read through the net. The best access time dropped to about 0.12 s when we tried to "parallelize" the access a little. Two users gave the AM command simultaneously, requesting status only half of the antenna each. But there were also trials when the access time of one or the other of the users could be very long, like 0.4 s, while the other still was about 0.1 s.

Maybe the pool-hosting Mac couldn't really cope with so many ports listening to external traffic. When we moved the ESHPOOL to the same host which run the simulated radar, so that all network access was over the loopback interface, the command to read the whole antenna took only about 80 ms. We are looking forward for repeating these tests in the actual E3D computing environment.

Box 16. Two users accesses one half of the array each via the Array Manager

We show the best cases out of several trials, there were drastic variations from trial to trial.

(a) Plain call (server doing vwaits)

```
esh(user1): at fm { tic; set D [resh uhf AM getstatus $d1]; toc -v }
14:08:00.001038 14:08:00.201980 ==> 0.200942
esh(user2): at fm+0.02 {tic; set D [resh uhf AM getstatus $d2]; toc -v}
14:08:00.020827 14:08:00.120638 ==> 0.099811
```

(b) Each user gets his own coroutine

```
esh(user1): at fm { tic; set D [resh uhf RtnAsync AM getstatus [list $d1]]; toc -v }
14:15:00.000866 14:15:00.128306 ==> 0.127440
esh(user2): at fm+0.02 {tic; set D [resh uhf RtnAsync AM getstatus [list $d2]]; toc -v}
14:15:00.020286 14:15:00.138098 ==> 0.117812
```

(c) Each user gets his own thread

```
esh(user1): at fm { tic; set D [resh uhf RtnAsyncTpool AM getstatus [list $d1]]; toc -v }
14:27:00.000984 14:27:00.112915 ==> 0.111931
&
esh(user2): at fm+0.02 {tic; set D [resh uhf RtnAsyncTpool AM getstatus [list $d2]]; toc
-v}
14:27:00.020860 14:27:00.117463 ==> 0.096603
```

6 Summary

Overlapped and intertwined with the specified Tasks of this work package, the basic aim has been make sure that the current-day Eros can be sufficiently strengthened to meet the requirements of E3D. In Eros5, there has been are some known problems: somewhat flaky context-switching, lag of support of multitasking and parallel processing especially on the user-level, and seriously non-optimal handling of long-living commands on the server side. Also some lingering doubts have persisted about the scalability of Eros to handle at least an order of magnitude more communication nodes than in the present systems.

In this report, we have addressed those points. We conclude that none of these potentially serious problems is a showstopper. Our work in this WP has been two-pronged. First, we have simplified Eros use, maintenance, and further development, by splitting Eros in two parts.

- The radar operating system EOS is the actual interface the radar resources, somewhat like a computer operating system provides coordinated access to the computer's resources.
- The e-shell (ESH) is for user interaction with the EOS, like running and monitoring experiments.

The second leg in our solution strategy has been to make much fuller use of the Tcl/Tk system's features and services than previously.

The following stuff is new in Eros6.

- Use of Tk to implement pop-up output windows for ESH for all kind of synchronous and asynchronous messages and notifications.
- Use of Tcl namespaces and multiple interpreters to increase modularity and code management.
- Use of coroutines and threads for multitasking and context switching.
- Use of two advanced features of the standard Tcl COMM IPC package. Coupled with thread pools and coroutines, these features have allowed us to significantly increase the efficiency of Eros IPC. In particular, we believe that our handling of long-lived commands is now about as good as it can be within this kind of system. If this is not sufficient, we need to start coding in C.

We have coded a working first implementation of ESH, with some 2000-3000 lines of new Tcl code, available as the package ESH in EISCAT HQ CVS repository. The EOS part of Eros6 is still mostly in development. For realistic testing of ESH, and the problematic points of Eros5 on the server side, we have made a temporary interface between ESH and Eros5. This interface allows ESH to talk with Eros5 server as if talking to Eros6 EOS. The 1200 line interface file is ESH_EROS5.TCL in the Eros5 LIB module, and is also available in the CVS repository. Some of ESH testing has been done at the EISCAT Heating site together with Mike Rietveld, as part of general Heating Eros development. We have tested ESH also at other mainland sites. Eros6 is not yet ready, but even in its present form, it should be possible to run all current experiments unmodified on ESH.

We have also taken some rudimentary first steps towards adding an E3D module to the Eros simulator, by coding initial tools for an "array manager" (AM) module. These allow convenient management of a pool of one hundred or so ESH-processes to represent the E3D array-group (AG) communication nodes.

7 Next steps

The aim is get Eros6 ready for operational use during spring 2014. Apart from further testing and documentation, the main work to that end is reorganize, strip down, and merge with esh_eros5.tcl, Eros5 server side so that it can serve as the first version of EOS. The Eros booting system will be overhauled at the same time. The screen footprint of EOS will be diminished. In Eros6, both users and operators normally should only see one or more ESHes on their screen. They are lightweight (boot time a fraction of a second), and will come and go as need be. The EOS will be always on, boot automatically when the computers boot, and will be mostly invisible.

On the way towards E3D, features will be added to the Eros simulator as the hardware configuration becomes better known. A decent Eros-level simulator for E3D should be available early. New E3D-specific Eros commands, experiments and maybe even practices can then be developed, tested, and iterated at the time when E3D hardware is starting to mushroom all over the northern lands.

The existing EISCAT radars will provide good testing and development environment for the many aspects of the E3D Eros that are common to all EISCAT systems. One of them is the need to maintain accurate system state. This has been somewhat neglected in Eros so far, but must be properly addressed for E3D. That is, the box labelled as “persistent system state” in Figure2 will need to be implemented.

On the current EISCAT systems, there are numerous programs, both graphical and terminal-based, that can, and often do, read from, and sometimes also write to, the radar hardware. At ESR, this all should happen strictly via EOS, otherwise, there will be no chance of the EOS of keeping accurate system state. It would seem to make a good exercise, from several points of view, to start integrating these various control and monitoring tools into Eros6.

APPENDIX Keeping track of net addresses of the e-shells

Unlike the EOS part of Eros, which has fixed, well-known (kept in configuration database) network address so that any sufficiently authorized ESH or EOS can connect to it, the ESH processes do not have well-known addresses. When an ESH client connect to EOS, a bidirectional communication channel is opened, and it may be is possible for the EOS to learn "something" about the net address of the connected ESH. In this Appendix, we inspect in detail what that something will be in various situations, and how it can be used to facilitate also ESH to ESH communication in some important cases.

The most important case is that any EOS and any ESH must be able to stop an experiment running in some particular ESH, call that ESH the ESH_EXP. Assume the ESH_EXP runs the experiment in hardware controlled by EOS at a remote site RAD (clearly at least one RAD; there can be more than one involved but that is an extra bonus). Even though we assume that typically the experiment would be controlled and stopped by the operator of the ESH_EXP, it must be possible in the very least for the operators at the E3DCORE site to stop that experiment. Similarly, a properly authorised ESH running anywhere in the web should also be able to do that. Because the experiment script is actually being executed in the ESH_EXP, which a priori has an unknown address, how can the relevant parties contact it so that the necessary stopping actions can be taken in that ESH? The solution is based on the observation that the EOS site (or sites) RAD which is used to run the experiment, has the bidirectional communication channel open to the ESH_EXP, and the end-point of that channel obviously must be known to the EOS/RAD.

Inspection of the source code of the Tcl COMM package showed that the package maintains the information about open channels in the state array of name COMM::COMM. Ideally, one wishes that the full network address (port number + IP address) of the channel would be somehow available. Were it so, then any ESH could just use a commands like

RESH *full_net_address_of_esh_exp* STOPEXP

to stop the experiment. But that turns out to be too much to hope. One cannot trust that the full address of the ESH_EXP is recorded anywhere where it would be accessible to Eros, see the case (4) below. But this turns out not being necessary either. The important thing is that a unique, persistent pointer to the ESH_EXP–to–RAD channel is maintained in COMM. Our strategy is to grab that pointer when the ESH_EXP first contacts the RAD to start the experiment. This pointer can be used instead of the true network address in RESH to access ESH_EXP. We save the pointer in an experiment-specific variable at RAD. The pointer is printed out by the PRINTEXP command on the RAD, thus is knowable to anybody who might need it. The connection between an experiment and the channel pointer should ultimately allow an experiment to be stopped transparently. In fact, in the E3D case when there can be multiple experiments running simultaneously at any given EOS, the COMM channel pointer of the controlling ESH could serve as an unique ID for the experiment itself.

We now describe in more detail how the above scheme is implemented.

The EOS keeps track of all connected EOS and ESH systems. For definiteness in the example, let's say that

- The EOS is a simulated SOD system running in the computer jofi at EISCAT SOD site. The simulated SOD listens in port SPORT.
- A "console user", ESH(sod), at port number CPORT is in jofi, with SOD as its "local radar" (default destination for the RR command).
- A "LAN user", ESH(lan) at port number LPORT is in the computer jini that has IP number IPJINI in the SGO LAN. This also has SOD defined as its local radar.
- A "WAN user", ESH(wan) at port number WPORT is in hserver computer at the EISCAT Heating site. This also has SOD defined as its local radar.

We now illustrate how the remoteID–to–socketID mapping maintained in the SOD COMM evolves when new clients connect to it using the RR or RESH commands. In the Eros6 implementation, the ENGINE process actually maintains two different COMM communication channels, called the REMCOM and ESHCOM channels. Each of these can support any number of individual socket channels. The REMCOM channel, inhered from Eros5, handles the receiving end of the RR commands. The ESHCOM is new in Eros6 and handles the receiving end of the RESH commands. Two channels are needed due to the slightly different protocols adapted for the RR and RESH commands. We only show the REMCOM connections here.

No connetions. With no remote connections to SOD, the mapping (the COMM "peers list") is empty.

Case 1. When the console ESH(sod) has connected to SOD by giving any RR sod command (but only then, the ESH–to–EOS connection does not appear simply when the ESH is launched), a peer appears in the SOD peers list as

comm::comm(REMCOM,peers,CPORT) = sock7fbc411b0290. (1a)

The console peer is identified by a port number (CPORT) and channel name (REMCOM).

The SOD EOS can now access the console ESH(sod) asynchronously by using RESH. For example, one could use

RESH -channel remcom CPORT NOTIFY -speak "hello there". (1b)

Other Eros6 systems can access ESH(sod) by going though SOD, as

RESH sod [list RESH -c remcom CPORT NOTIFY -speak "hello there"] (1c)

Case 2. When the ESH(lan) connects to SOD, the peers list is augmented by the entry

comm::comm(REMCOM,peers,LPORT IPJINI) = sock7fbc411b0e10 (2a)

Picking the fully qualified address from the peers list, any EOS and any ESH could access ESH(lan) using

RESH { LPORT IPJINI } *command* (2b)

Case 3. What happens if ESH(wan) in hserver in the WAN uses, for enhanced security, an SSH tunnel to the SOD host jofi, so that it can connect to SOD using localhost? The tunnel has been set up with

SSH -L SPORT:localhost:SPORT jofi. (3a)

The ESH(wan) now connects to SOD using RR to localhost:SPORT, as e.g.

RR {SPORT localhost} PRINTEXP. (3b)

The peers list is augmented by

comm::comm(::REMCOM,peers,WPORT) = sock7fbc411b0f10. (3c)

There is no indication in the peers list entry that the "REMCOM WPORT" socket's other end is ultimately in the WAN. Nevertheless, COMM and SSH work together so that the ESH(wan) in the WAN can still be accessed from SOD in a similar way as in (1b), using

RESH -channel remcom WPORT *command* (3d)

For instance, to query the hostname of ESH(wan), one could type the following in the SOD console ESH(sod)

```
tic; puts [resh -channel remcom WPORT info hostname]; toc          (3e)
==>
hserver
0.023466
```

The output shows that the remote command from jofi in SOD to hserver in HEA took about 23 ms to return.

In Case 3, due to the SSH tunnel to localhost, the fully qualified address of the ESH(wan) in the WAN was not available even while one could access the SOD peers list. Nevertheless, anybody who wants to send commands to ESH(wan) can in principle go through the SOD EOS by using the appropriate peers entry "REMCOM WPORT" to pick the right socket. For instance, the ESH(lan) in the SOD LAN can chain the RR and RESH commands as

```
RR sod RESH -channel remcom WPORT INFO HOSTNAME                    (3f)
==>
hserver
```

to execute the remote command INFO HOSTNAME in the hserver at HEA. In actual practice, one still needs to know that it is this particular label, "REMCOM WPORT", which one should be looking for in the peers list. This requires extra bookkeeping by Eros6, in addition of what COMM does automatically.

The most common case when one needs to find out how to address a remote ESH, let it now be the ESH(wan), is when one wants to stop an experiment running in ESH(wan) and accessing a known radar, now SOD. One wants to use one's own ESH, now ESH(lan), to give the STOPEXP command to ESH(wan). The required addressing information is in practice found by giving the command PRINTEXP to SOD from ESH(lan). We will describe in detail how that information is made available in Eros6, but first a little background about running experiments in Eros6.

With Eros6, it is possible to run an experiment either in the Eros5 way, that is, in the EXPDOER process of the target RAD, or in an ESH process anywhere. The latter requires that the ESH has the RAD defined as its default destination radar. The standard Eros5 command to start an experiment in the EXPDOER process is

RUNEXPERIMENT *elanfile starttime parameter*...

In Eros5, the RUNEXPERIMENT command typically is given either at the RAD console, or from the Eros console of some other radar site, using RR. When an experiment is started in that old way in Eros6, there is no problem in giving the necessary stop command: one just executes the standard Eros5 STOPEXPERIMENT command in the target system, either by giving it locally in the target system console, or by sending a remote command to the known address of the RAD.

The new feature of Eros6 is that it is possible to run experiments in an ESH process anywhere in the web. In practice, security issues might come into play and restrictions might be put in place.

To start a SOD experiment file locally in the ESH(wan), one uses an ESH command with the same command name, RUNEXPERIMENT, as in Eros5. When that command is given in ESH, the *elanfile* is sourced in the ESH, and in addition, an "info command" is sent to SOD for administrative purposes.

There is a complication that we mention here, even though it is not strictly relevant to the main theme of this Appendix. The *elanfile* will typically refer to a multitude of files to be loaded into radar hardware. Some of these files are system files, kept in fixed locations at the RAD, while others are more or less fully under the user's control. In the initial Eros6 implementation, all experiment files referred to in the *elanfile* need to be available at the RAD site, with the names used in the *elanfile*. There is considerable opportunity for confusion if the file hierarchies in the computers running the ESH and the EOS are different. The long-term plan is to use the virtual file system machinery (TclVFS) of Tcl to provide a uniform file name environment for experiment-related files in EOS (a VFS would also be useful for the EOS simulator). Another possibility would be to change the Eros system commands that load files to hardware so that they transfer the files from the user's computer at load-time.

We now describe how the info command is implemented in order to ultimately make the effective *remoteid* of ESH(wan) available via the PRINTEXPERIMENT command.

The info command that actually is sent to the RAD for remote execution is ESH::EROS5::RUNEXPERIMENT_2, defined in eros/lib/esh_erost5.tcl. It has the header line

```
proc RUNEXPERIMENT_2 {remoteid expfile startsec args } { ...
```

At the time when the RUNEXPERIMENT_2 command is *executed* in the ENGINE process of the RAD, the first parameter will be the port number WPORT that identifies the socket channel's receiving end in the RAD peers list. But while the other parameters in the parameter list have precisely the same values at execution time as are set at the time when the remote command is sent out, the first parameter is different. As will be shown, at the *call time* of the info command, in ESH(wan), the first parameter has the string value "__IPID__". This value will converted to the needed numerical value behind the scenes before the remote command is actually executed in RAD.

One of the administrative tasks of the ESH::EROS5::RUNEXPERIMENT_2 command is to make available to Eros users the ESH–to–RAD socket channel's identifier label. This requires some behind-the-scenes cooperation by the COMM packet's internal routines. Note that the client ESH itself cannot, in general, provide its own IP address to the RAD server (as an explicit parameter in the info command, say). This is because a Unix process does not normally know its own IP address.

Thus, instead of trying to uniquely identify the ESH–to–RAD channel based on the sending end, the channel identifier is formed based on facilities at the receiving end. That end anyway is what is really of concern to the external word that needs to communicate with the ESH.

To grab the channel identifier, which the Tcl COMM calls the *remoteid*, a piece of code needs to be added to the plain, out-of-the-package, COMM system. The COMM package provides a hook for the programmer to provide the required piece of code. The hook is executed immediately after the COMM channel has collected a full Tcl command from the receiving socket, but before the received command is yet given for the Tcl interpreter for execution (security authentication can also be done at this time if required). At that time and place, but not necessarily later, the appropriate *remoteid* is readily available *and* it is also known to what incoming client *command* that id belongs to. Thus the hook can pair the *remoteid* with the *command*, so that the id can be used when the *command* is, maybe much later, finally executed. In more detail, the following happens.

In Eros6, the hook (routine EshcomHook in eros/lib/esh_eros5.tcl) simply appends the *remoteid* to the *command* and then calls a separate routine, EshComRec (also defined in esh_erost5.tcl), to actually decide what to do with the *command*. It depends on the received *command* whether the *remoteid* is ultimately needed or not. Therefore EshComRec needs a way to decide when it should actually propagate the *remoteid* forward to the executed command. The present implementation is such that on the client side (which, as stated, does not know the relevant remote id), the macro name, or keyword, __IPID__, is placed in the position in the *command* where the actual *remoteid* should go. EshComRec uses regsub to replace the string __IPID__, if present, with the actual *remoteid*, and then finally executes the remote command that the client requested.

The hook processing needed to grab the *remoteid* of the sending ESH and save it as one of the administrative parameters of the experiment, happens transparently to the user. What is visible for users is an extra line labelled as "Id" in the standard Eros5 PRINTEXPERIMENT command output. In Eros6, the existence of this line is the sign that the experiment script mentioned in the printout is actually not running in the EOS locally, but is running in a remote ESH. There are actually two parameters in the Id line. The first is the *uid* of the ESH, and the second is the *remoteid* of the ESH–to–RAD channel, as seen in the receiving end. The *remoteid* is needed to access the remote ESH. The *uid* is meant to label an EISCAT user uniquely, so that one can e.g. send e-mail or make a call based on that info. But it will not normally label an ESH uniquely.

When the SOD experiment *manda* is running in the ESH(wan), the PRINTEXP command at SOD would show something like

```
EXPERIMENT (SOD) 18-Jan-2014 07:02:22.9
---------------------------------------
Id          : wan WPORT
Exp file    : manda.elan.ecod
    dir     : /kst/exp/manda
    state   : RUNNING since 18-Jan 07:02:09.7
    ...     : ...
```

The Id line contains the *uid* "wan" and the *remoteid* WPORT of the ESH that runs the experiment. We do not show the COMM channel that appears in the COMM peers table, for it will always be REMCOM when the experiment is running in an ESH. The command that other ESHes can use to stop the experiment in the ESH(wan) is sent via SOD, using

RESH sod RESH -channel remcom WPORT STOPEXP.

For operational use, wrappers need to be implemented to spare users from details like the option "–channel remcom".

There is a potential problem in the way COMM maps the *remoteids* to socket ids via the peers list. What happens if in (1b) and (3d) the port numbers CPORT and WPORT are identical? They can be identical because they refer to different computers. If we would not use tunnelling to localhost, the *remoteids* would differ by the host IP number, so the *remoteid*s would be unambiguous. But that is not the case now. When CPORT=WPORT, the commands (1b) and (3d) become identical, so they obviously cannot reach to two different clients.

Testing showed that the connection that was established first always prevails, and therefore the second client cannot be reached using RESH *remoteid* asynchronously. It seems that after COMM has created the peers mapping for a particular value of *remoteid*, it does not update the peers to socket map when (only) the right-hand-side of the map changes. Further inspection and testing showed that the two COMM socket channels themselves nevertheless are OK so that both ESH(sod) and ESH(wan) can send synchronous commands and get replies from them normally. That works because internally in that situation COMM does not actually make use of the peers-to-socket map, but works directly with the correct sockets.

But for our purpose, we need to be able to use the correct socket also "asynchronously". A work-around to the ambiguous peers–to–socket map, at least when stopping an experiment, is to use a COMM hook to pick the actual *socketid* at the time the initial RUNEXP command is received, in the same way as the *remoteid* is picked. Then one forces (temporarily only, to be on the safe side) that *socketid* into the right-hand-side of the peers-to-socket map just before the experiment stopping RESH command is used. That works OK for stopping an experiment with RESH, but is not a general solution to the ambiguity problem. Another work-around is to avoid the possibility of ambiguities in the first place. For example, one could discourage tunnelling like in (3a), but that surely is not the ideal solution either.