Data Acquisition Interface Design

Extending the NASA VLBI Field System

Version: 25.6.2009

Helge Rottmann[1], Alexander Neidhardt[2],
Martin Ettl[2] , Christian Plötz[3],
Ed Himwich[4]

[1] Max-Planck-Institut für Radioastronomie, Bonn

[2] Forschungseinrichtung Satellitengeodäsie
Technische Universität München
Geodätsches Observatorium Wettzell

[3] Bundesamt für Kartographie und Geodäsie
Geodätisches Observatorium Wettzell

[4] NASA/GSFC/NVI

Table of contents

1 Introduction

The requirements for VLBI data acquisition systems are constantly increasing. A higher observation density, real-time access for more dynamic schedules, highly automated observations and remote control of complete sites for example will offer new possibilities but will also require new add-ons for the controlling software. To improve the situation (semi-) autonomous, remotely accessible control systems must be introduced. Such complex systems require reliable, transparent and modular structures from the upper controlling layers down to the basic single components in combination with sophisticated safety mechanisms for automation.

This report will sketch the path for extending the current NASA VLBI field system in order to serve the requirements mentioned above. The ideas presented here are based on experience gathered with a working prototype system operated at the German Geodetic Observatory in Wettzell, one of the geodetic Fundamentalstations (see eg. [NEID06] and [NEID09])

2 VLBI data acquisition – the current state

At present the VLBI data acquisition is controlled by the NASA VLBI Field System (hereafter FS). The FS consists of a collection of memory resident programs (see [FSLink]). Communication between the individual processes is handled via a common shared memory segment and the use of semaphores. A sketch of the FS architecture can be seen in Fig. 1. The FS is a very stable, well known and well-supported system. However, in its current state it is unable to cope with some of the new data-acquisition requirements. The most crucial limitation at present is the lack of remote-control capability and the inability to export system status information to non-local users and services. Also, the restricted standardization of native TCP/UDP support and the fairly high complexity of the “historically grown” FS architecture make the task of integrating new data acquisition hardware a challenging process.

Native TCP/IP and UDP support

The data acquisition equipment at a typical radio observatory consists of numerous, independent hardware components each for a single specific task. The connection of these components was in the past typically realized via serial or parallel buses (eg. MAT, GPIB). Only recently has communication over local ethernet networks been utilized (at present mainly by the Mk5 unit). While the FS provides support for all serial/parallel line communication specifications, currently there is only limited native FS support for ethernet-based communication. However, many new hardware components (e.g. dBBC, DBE etc.) will use TCP/IP or UDP over ethernet for hardware control. At present, for every new Ethernet device the corresponding FS control module would have to fully implement TCP/IP or UDP socket communication functionality. This is a crucial limitation of the existing FS for several reasons:

Apart from the significant and redundant work imposed on the developer this is not an optimal procedure in terms of system stability. Especially the TCP/IP protocol – when not cleanly implemented on both sides of the interface – can cause socket hangs and would require restarting the relevant FS components. In fact, because of problems in the Mk5 unit's TCP/IP support, this sometimes happens at the stations, causing data loss until noticed and fixed by the station operators. A solution for generating modular, robust and reusable ethernet based communication interfaces is provided by the *idl2rpc.pl* middleware generator developed by the geodetic VLBI group at Wettzell (see section 3.2 for details).

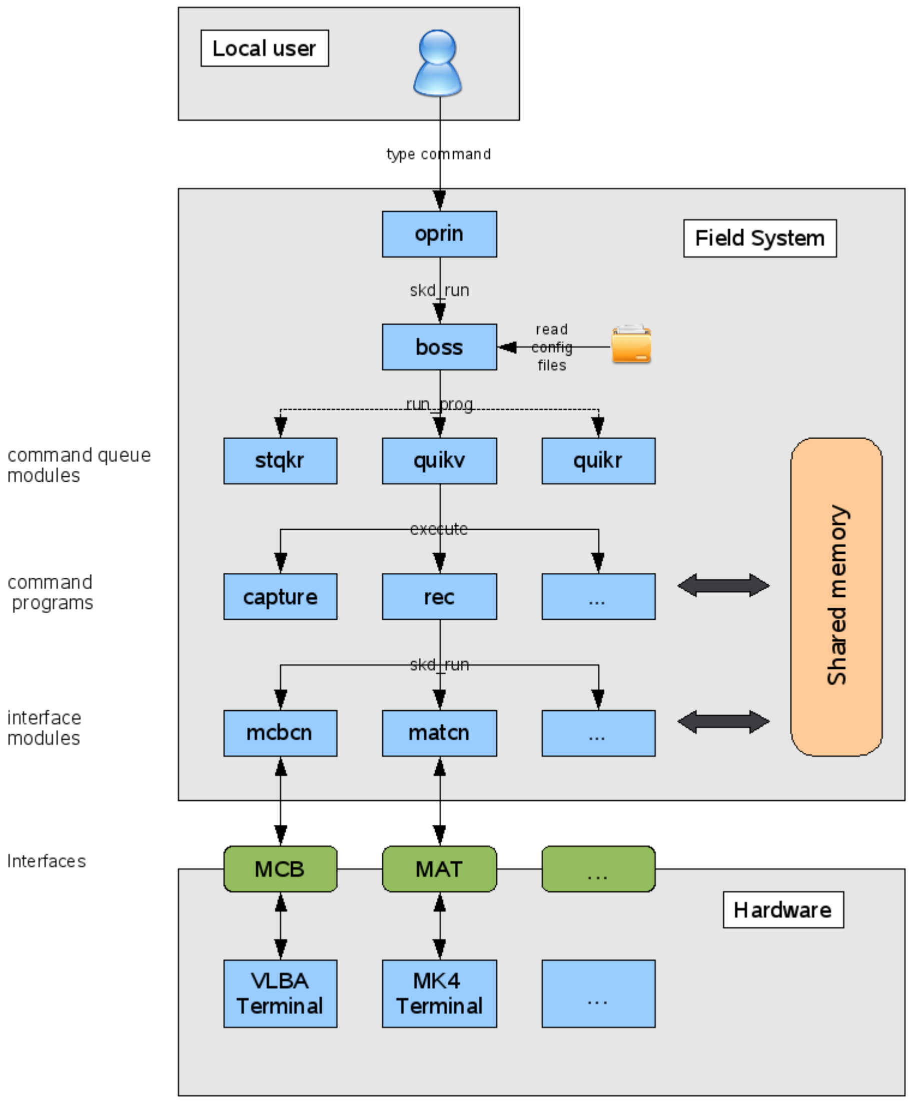

Figure 1: Sketch of the FS architecture. Each VLBI station hosts a local installation of the FS software with a localized set of configuration files (describing for example connected hardware etc.). The FS accepts console input from a local user or alternatively can process batch files containing SNAP commands.

Native Remote Control

The FS presently has only very limited built-in remote control capabilities. Input commands can be issued to the FS locally, either interactively by the operator through the *oprin* console (see fig. 1) or by supplying batch command files. The FS response is displayed locally via terminal windows. The "inject_snap" command allows simple commands to be given remotely. While local operation is a feasible scenario for most situations, there are cases where remote control is desirable. For very remote stations (e.g. the German Antarctic Receiving Station (GARS) O'Higgins which is operated by the Deutsche Zentrum für Luft und Raumfahrt (DLR) and the BKG) the logistic effort for manned operations is high. X-window forwarding or VNC is often not feasible due to the very limited bandwidth available (e.g. about 256Kbit to o'Higgings in the current setup). On the other hand eVLBI with real-time correlation would require at least some control of the correlator over the observing station in order to obtain calibration and other status information. Grid-based correlation might also stand in need of a mechanism to dynamically direct the output datastreams from the station to the various grid nodes.

Extendibility / Complexity

The complexity of the FS architecture is fairly high and public documentation is out of date. While the FS architecture is (to some extent) modular, integrating new hardware into the system still requires code and configuration changes in several places. Enabling new commands for existing hardware would for example require changes in at least three different locations. One of the goals for a FS extension would be to reduce complexity and make the code more modular.

3 Extending the field system

3.1 Design requirements

As outlined in the previous section an extended FS should natively support ethernet-based communication, allow for remote control and increase the general maintainability and extensibility. With respect to the architecture there are a few more crucial requirements:

Simplicity

Due to limited financial resources and IT support at some stations we regard simplicity as a crucial design requirement. The proposed solution should not greatly increase the system's complexity or demand for powerful hardware. We also regard the utilization of modern middleware solutions (e.g. Webservices) to be unfeasible. Despite their usefulness for creating distributed systems, these technologies require special infrastructure (e.g. application servers) to be set up and maintained which might be an impassable barrier for some stations. Also, the demands on the station's firewall configuration should be low. Ideally, the system should require only one (configurable) open port.

Long term maintainability

Software solutions for VLBI typically have a long lifetime. The FS has been in use for ~30 years. The choice of technology for the FS extensions must consider such high lifetime demands. We advocate using well-established, well-supported and generally available technologies e.g. ONC-RPC for the communication layer (see Section 3.2).

Integration of the FS

We suggest an extension of the FS, not a rewrite. In fact the FS should become fully integrated into the new architecture without any changes to the FS code. Instead of altering the FS code, we suggest to place all new functionality in an extension layer above the FS (see section 3.3).

3.2 The communication layer

The interaction between the data acquisition system and the individual devices will be realized in a client-server model on the basis of the TCP/IP or UDP protocols. The communication will be implemented using standard remote procedure calls (RPC). From a programmer's point of view calling an RPC is identical to calling a local procedure. The client has no knowledge of the processing end-point of the procedure call, handling of the control and data flow is realized by the RPC layer (Fig. 2).

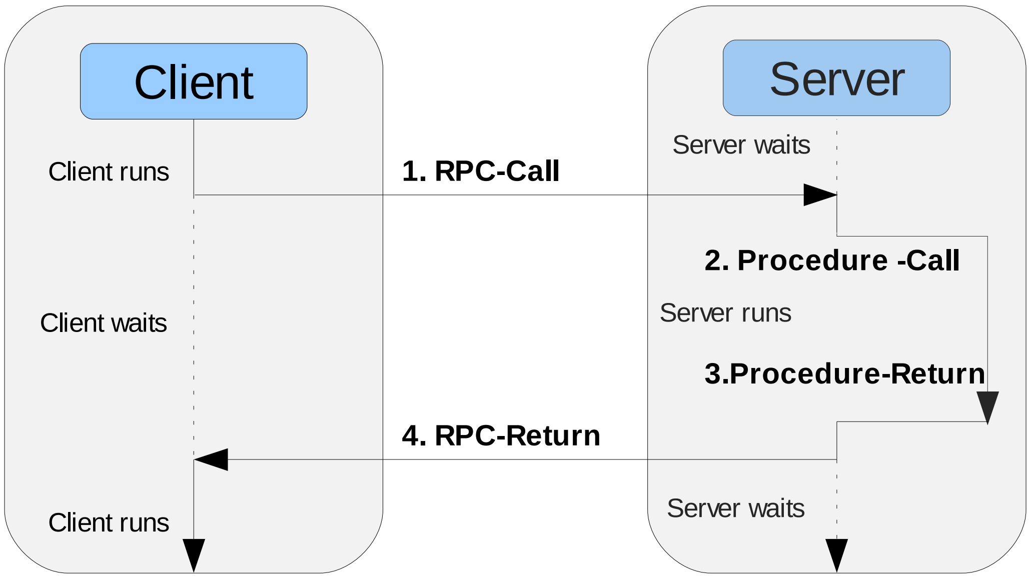

Figure 2: Sketch of the RPC client/server model.

To reduce the effort of implementing the communication layer an RPC middleware generator has been developed by the Wettzell group (see [NEID08], for basic middleware structures also see [NEID06]). The generator (named i*dl2rpc.pl*) automatically creates the entire communication layer starting from an interface definition file *(.idl)*. The interface file contains an abstract (but easily understandable, C-like) description of the communication interface between the client and the server (an example for a simple .idl file is displayed in Fig. 3).

The generator script creates a number of C++ code modules, an additional output option for pure C code (for direct integration into the FS) is planned. The application developer will make code changes only in two places: in the client source (to be written by the developer) and in the server file skeleton provided by the *idl2rpc.pl* generator. These two files will contain the actual applications logic and device specific implementations.

The communication between client and server is realized via a single (configurable) port. The default protocol can be configured to be either TCP/IP or UDP (the utilized protocol can however be

```
interface fsmc
{
       void vReset();
       unsigned int uiGetSystemStatusMonitorText (out string strStatusTags <>);
       unsigned int uiGetSystemOverallStateText (out string strStatusTags <>,
                                                out string strTempTags <>,
                                                out string strMark5Tags <> ,
                                                in unsigned long ulLogDescriptor,
                                                out string strLogText,
                                                out string strAdditionalLogText);
}
```

Figure 3: Example for a simple interface definition file that serves as input for the idl2rpc.pl generator script. The syntax is very much C-like. Input and output parameters are marked by the "in" and "out" keyword respectively. For a full description of the interface definition language used by the idl2rpc.pl generator see [NEID08] .

switched at runtime if desired). The server is automatically controlled by a watchdog process which restarts the server and allows the client to immediately reestablish communication to the server after unexpected crashes or communication failures. In addition the code created by *idl2rpc.pl* allows executing server functionality in a parallel periodic loop.

The generator uses the ONC-RPC (Open Network Computing Remote Procedure Call) implementation and the *rpcgen* program to create the RPC communication layer. Both are well-tested and supported for several decades and are available on nearly every Linux system.

In summary the i*dl2rpc.pl* generator provides a very easy, yet powerful mechanism to design a distributed system consisting of several independent servers which act completely autonomously and communicate with each other via a standardized communication layer. This allows splitting a complex, monolithic architecture like the FS into several manageable units. Each individual unit can be developed, tested and operated also outside the FS. Different clients (console based, GUI-based, browser-based etc.) can be developed which all interact with the same server component.

3.3 The extension layer – a first approach

As outlined in section 3.1 a FS extension should leave the existing FS intact and rather integrate it into the new architecture. In a first approach it is therefore necessary to add an extension layer that is located above the FS core and which will permit remote control of the FS. The architecture of such a basic system is sketched in Fig. 4.

The interface server serves as the central (single port) service to which remote clients can connect

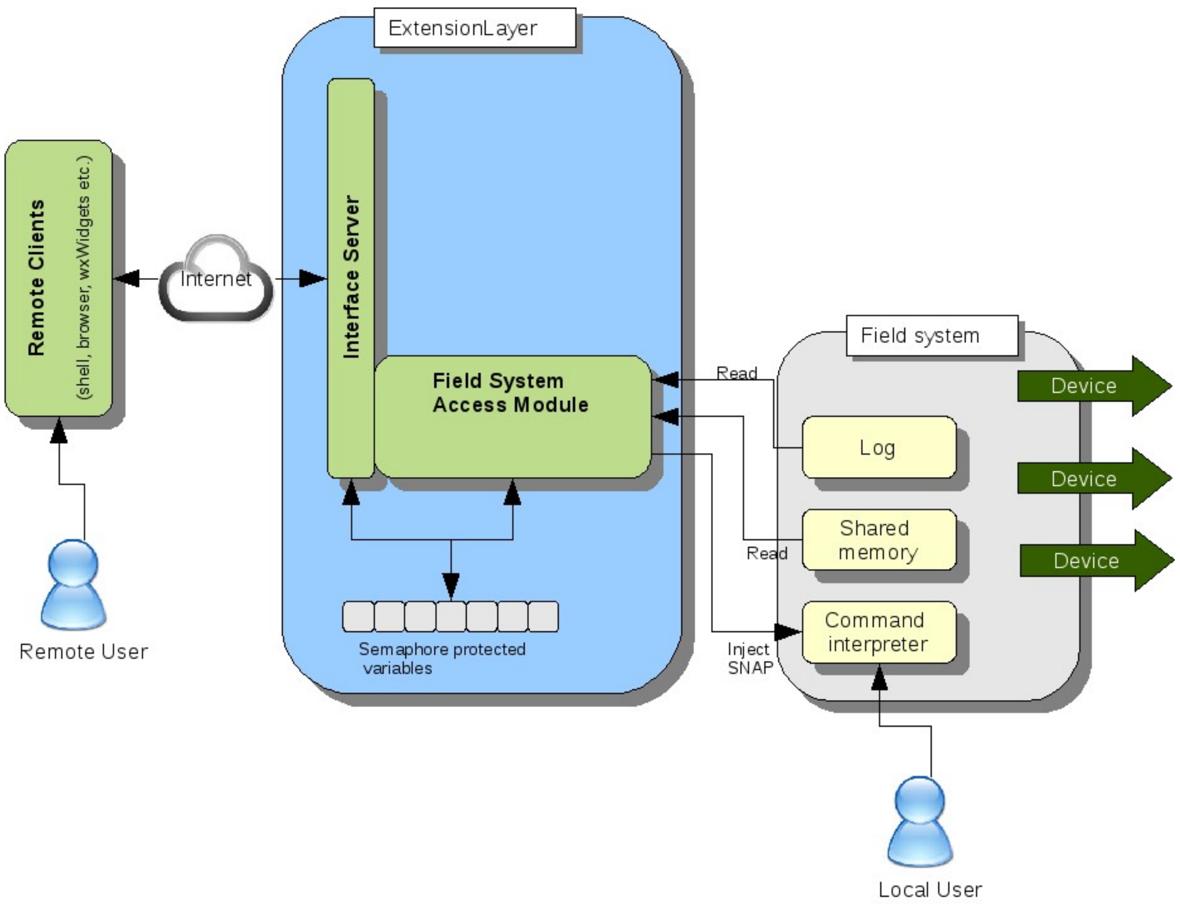

Figure 4: Sketch of the basic FS extension architecture. Remote clients can control the FS via a RPC interface server. The server periodically queries the FS's log and it's shared memory contents and exposes these via RPC. Also, the clients can issue SNAP commands to the interface server, which passes these on to the FS via SNAP injection.

for remote FS control. In a first step the interface server will simply consist of the "field system access module". This component reads the FS's logging information and the shared memory contents and exposes these via RPC to remote clients. In addition the server accepts SNAP commands and simply passes them on into the FS via SNAP injection. Semaphore protected variables are used to serialize concurrent client or server-loop requests.

3.4 The extension layer – adding device control

In the basic state of the FS extension as outlined in the previous section the device control would be handled by the FS in the traditional way. For new devices (e.g. the DBBC) it would be possible to implement the device specific communication layer using the *idl2rpc.pl* generator described in section 3.2 and then compile the client code into the FS (compare Fig. 1). In addition the commands to be accepted by the hardware would need to be added to the command queue modules (e.g. *quikv*) and registered in the configuration file (*fscmd.ctl*).

To make integrating new hardware an easier task and to open the path for a more modular FS architecture in the future we strongly advocate to add device control capabilities also to the extension layer (see Fig. 5). This will however, require to replace some of the control elements in the current FS control loop to the extension layer control loop (e.g. interpretation of command batch files etc). For a smooth data exchange between both a bidirectional control communication is desirable.

In order to be able to control the devices, some form of command interpretation needs to be added to the extension layer. Since the functions accepted by the various devices have already been defined in their interface idl files, the *idl2rpc.pl* generator could be extended to automatically create

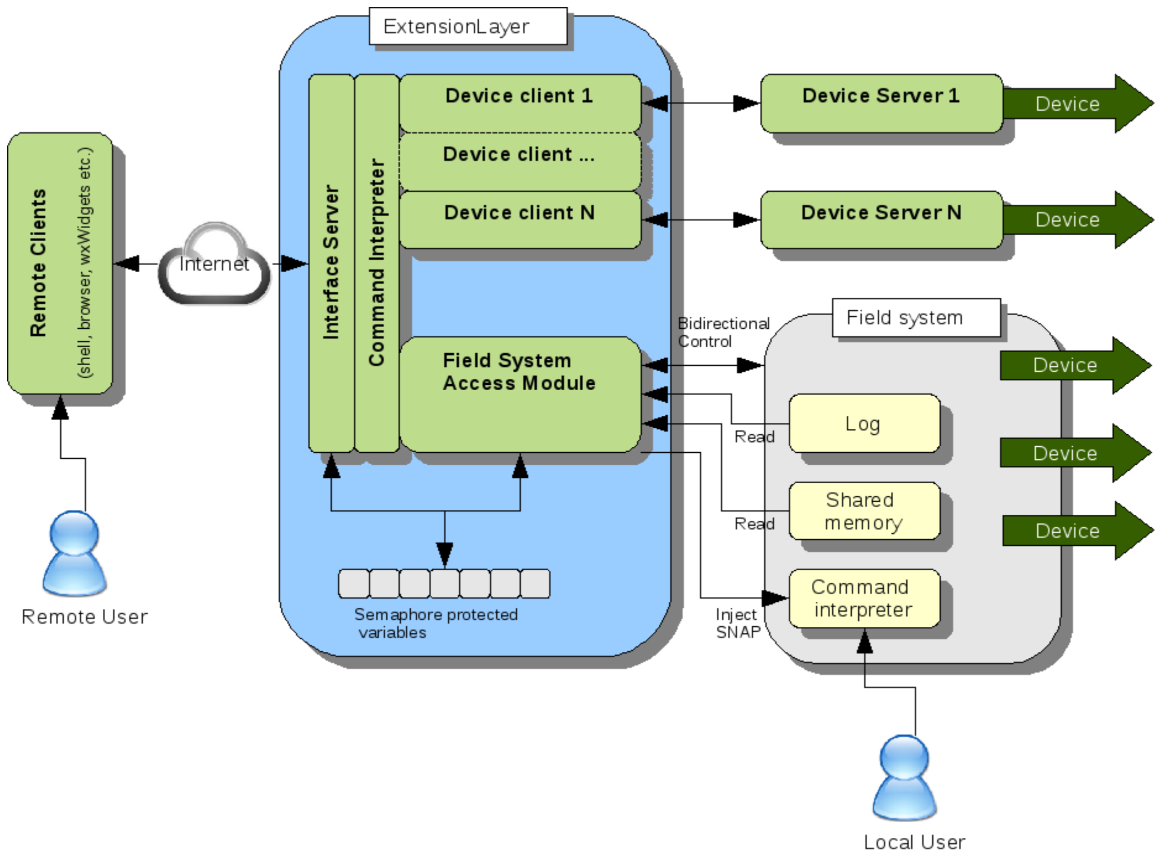

Figure 5: The extended FS with device control added to the extension layer.

command parsing code for each client using the same definitions. In a first approach the accepted commands could be identical to the defined RPC procedure names. Mapping these RPC procedures to more meaningful command names would however be desirable. Also, very likely only a subset of all RPC procedures known by the server need to be exposed as user commands. This solution is attractive because all information describing the specific device will be kept in one central file (the interface .idl file) and all device control and communication code will be created by the *idl2rpc.pl* script.

When the frontend interface server receives a command, the command interpreter module would pass the command to the connected device clients. Each client then parses the command to determine if it is part of its own command set. If so, the command will be passed to the dedicated device server where it will be executed. Commands not accepted by any of the connected device clients will be passed on to the FS. Batch processing of *schedule* and *proc* directives will be handled in a similar way. Every command encountered in the batch queue will be passed to the command interpreter. If not matched the command will be passed to the FS.

3.5 Authorization and Authentication

The FS extension layer exposes FS functionality via RPC to remote locations. The RPC layer provides no intrinsic access control mechanism. Therefore every person or process that is able to reach the RPC server would be able to remote control the FS, which is obviously unacceptable for the stations. The ability to limit remote control to authorized persons or services and possibly also certain time periods (e.g. transient and/or eVLBI observations) is a crucial requirement for the FS extension.

In a first approach SSH tunneling can be used for authorization. The only requirements would be for the remote user/service to have a local user account and a valid public/private key pair. Because the client/server communication generated by *idl2rpc.pl* uses TCP/IP and UDP protocols over a single, configurable port, setup of such a tunnel is rather easy without the need to reconfigure firewall settings.

SSH tunneling has 2 major drawbacks:

SSH provides no authorization mechanisms. Once authenticated a user or service has full control over the remote FS. Optimally the FS extension should provide a more fine grained authorization scheme. For example, multiple users and services could be allowed to monitor the FS, while remote control should be limited to one single user. Possibly, role-based authorization would also be desirable, to limit parts of the RPC interface to certain groups of users.

The ssh tunnel is not controlled by the communication layer, and has to be established by the user prior to issuing any RPC client/server interaction. This can be problematic if the tunnel breaks down due to temporary failures in the communication line. In that the user would have to first reestablish the tunnel before the RPC communication could be resumed. A possible workaround would be a parallel process or script that monitors the state of the tunnel and automatically restarts it in case it has collapsed (however the problem of ssh not allowing password inputs from the standard input device needs to be overcome. Solutions exist in the literature.)

Thus, some form of authentication has to be build into the FS extension layer. Prior to executing a command received by the interface server the authentication module would check against a rule set whether the person or service has the appropriate rights to issue the particular command. The rule set should be configurable on the fly (e.g. via editing a simple configuration file) – without recompiling the FS extension layer. The rule syntax would allow to grant or deny access to RPC interface methods for certain users (and possibly also user roles) in particular time intervals.

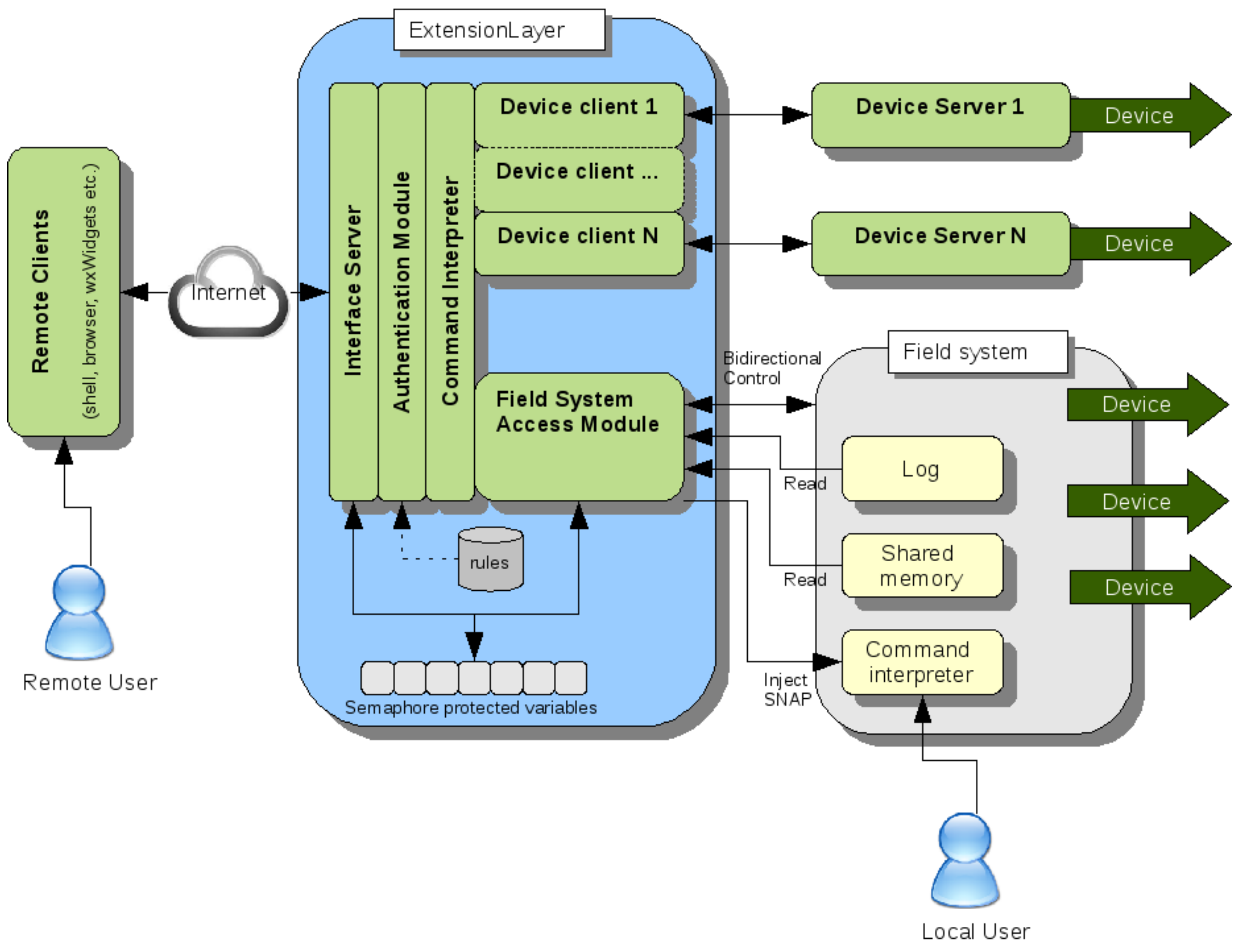

Figure 6: Sketch of the FS extension containing an authentication layer. Every RPC command will be passed to the authentication component prior to execution. The authentication layer consults its rule set to check if the user or service has the appropriate rights.

4 References

[FSLink]	Mark IV Field System Documentation, http://lupus.gsfc.nasa.gov/fsdoc/fshome.html
[NEID06]	Neidhardt, Alexander, Verbesserung des Datenmanagements in inhomogenen Rechnernetzen geodätischer Messeinrichtungen auf der Basis von Middleware und Dateisystemen am Beispiel der Fundamentalstation Wettzell, Mitteilungen des Bundesamtes für Kartographie und Geodäsie, Nr. 37, Bonifatius GmbH, 2006
[NEID08]	Neidhardt, Alexander, Manual for the remote procedure call generator "idl2rpc.pl", Geodetic Observatory Wettzell, 2008
[NEID09]	Concepts for remote control of VLBI-telescopes and for the Integration of new VLBI2010 Devices into the Field System, Talk presentation, http://www.fs.wettzell.de/veranstaltungen/vlbi/frff2009/