The following paper was originally published in the
Proceedings of the Twelfth Systems Administration Conference (LISA '98)
Boston, Massachusetts, December 6-11, 1998

# A Configuration Distribution System for Heterogeneous Networks

Gledson Elias de Silveira, Federal University of Rio Grande do Norte
Fabio Q.B. da Silva, Federal University of Pernambuco

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

# A Configuration Distribution System for Heterogeneous Networks

*Glêdson Elias da Silveira* – Federal University of Rio Grande do Norte
*Fabio Q. B. da Silva* – Federal University of Pernambuco

## ABSTRACT

This article presents a configuration distribution system that assists system administrators with the tasks of host and service installation, configuration and crash recovery on large and heterogeneous networks. The objective of this article is twofold. First, to introduce the system's modular architecture. Second, to describe the platform independent protocol designed to support fast and reliable configuration propagation.

### Introduction

Service configuration is one of the most common tasks for the system administrator. It is also one of the most critical, since it impacts network performance, resilience, predictability, and security. This task becomes increasingly difficult as the size and heterogeneity of the networks increase. As pointed out in [1], some of the reasons for this increase in complexity are:

- configuration of each service must be uniform over the network, requiring update on every host anytime a configuration change is required;
- in general, there are great differences in the actual format and location of the configuration files of a service for each platform. Therefore, to configure a service in a heterogeneous network amounts to configure the service in each platform;
- each operating system provides its own set of non-standard configuration parameters for the common network services. In most cases, it is necessary to learn and use these non-standard features to achieve optimal performance of the service in each platform;
- usually, large networks are managed by a team of system administrators. Configuration rules and parameters must be effectively communicated among team members to avoid inconsistencies that can arise due to personal preferences during the configuration process.

Several works have proposed methods and tools to assist service configuration and management [1,2,3,4,5,6,7]. A common underlying characteristic of those works is the existence of a (central or distributed) repository of configurations. The possibilities of configuration consistency checking and easy recovery of host configuration after a serious system's crash are just a few of the advantages of having a configuration repository. In [1], the repository holds configuration for various services and allows consistency analysis to be performed among different services.

Once service configurations are stored in a repository or database, they must be propagated to target hosts on the network. This article addresses the problem of distributing service configuration from a (possibly distributed) database to network hosts. It presents a configuration distribution system and a propagation protocol for configuration distribution in large heterogeneous networks. Hereafter, the system and protocol are referred to as CDS.

The following section provides an overview of configuration management and establishes the scope of the work presented in this article. Later, related work is compared to the approach of this article, and the CDS modular architecture is presented. Then, the propagation protocol is described in depth and some details of the CDS implementation are provided. Finally, some conclusions are presented.

### An Overview of Configuration Management

The Network File System (NFS) [8] is a widely used service that provides file sharing among hosts in the network. NFS is used here to illustrate the problem of configuration management. The following steps are usually performed to manually configure the NFS service:[1]

1. Login on the server and edit the NFS configuration file to export the desired filesystems to client hosts.
2. Initialize the NFS daemons on the server.
3. Import desired filesystems on the client, either manually or by editing the configuration files.
4. Under demand, unmount filesystems on the client side.

This manual process only works for small networks or in situations in which changes in the configuration of servers and clients are seldom necessary. However, even in those situations system administrators prefer to use some automated support to avoid inconsistencies and insecurities that may arise from

[1]The configuration of most services uses similar sequence of steps.

human error. Furthermore, this support can be critical during crash recovery, when a machine must be promptly reconfigured to its state prior to the crash. Ideally, to configure the NFS on large and critical systems, the administrator should perform a number of tasks in a coherent and consistent form:

- **Service planning**: to define the file systems exported by the servers and imported by the clients, as well as, their access and security characteristics. This task should abstract away from platform specific features.
- **Configuration consistency checking**: the planning of the service must be carefully checked for consistency. For instance, it should not be possible to plan a system in which clients import a filesystem that is not exported by any server.
- **Generation of configuration files**: from a consistent service plan, the NFS export and import files should be generated. At this stage, platform specific features must be used to create files on the right format for each supported operating system and hardware platform.
- **Configuration propagation**: the export and import files must then be distributed to the corresponding hosts.
- **Configuration activation**: the necessary actions must be performed on each host to activate the new configuration. It may be necessary to reboot the host.

The CDS supports configuration propagation and activation, and provides a platform to be used by any system implementing the task of configuration planning, consistency checking and generation of configuration files. A system that supports these tasks is fully described in [9]. This system uses the CDS as its configuration propagation and activation mechanism.

### A Comparison with Related Approaches

Recently, several systems were developed to assist the task of host/service configuration and management [2,3,4,5,6,7,10]. Generally, these solutions were designed to resolve local requirements. However, their technologies identify and evaluate various characteristics of the configuration process. There are many ways to classify and evaluate configuration distribution systems. This section addresses and compares the main techniques used to distribute and activate service configuration from a database to network hosts.

The configuration propagation can be performed using several distribution mechanisms. Other works [2,3,4,5,6,7,10] have used the following approaches: remote commands, TCP/IP socket, NIS, NFS or a specific protocol developed using the Remote Procedure Call (RPC) mechanism.

In UNIX systems, the *rcp*, *rsh*, and *rdist* commands can be used to allow remote copy and processing. The main drawback of r-commands is the need for privileged access on the remote hosts, which is a major security drawback. *Config* [4] uses *rdist* to distribute the configuration files from the repository to the network hosts. This repository is created in a server host as hierarchical directories, which can be replicated in other hosts. OMNICONF [6] also uses r-commands (in particular *rsh*) to manipulate files and directories that keep the configuration of the hosts in a central database.

Mechanisms based in the client/server model can be developed using the socket interface presented in TCP/IP environments. Its disadvantage is the requirement to manipulate the data formats of each platform. The main advantage is to require only the TCP/IP protocols running for using the solution. In addition, they do not demand privileged access on the remote hosts. However, it is very difficult and complex to work directly using the socket interface.

Other approaches have used a combination of NIS and NFS to distribute configuration information. The lack of an incremental update mechanism causes serious performance and network-traffic problems when using NIS in large networks. Moreover, NIS has notorious security problems. Furthermore, to use these distribution mechanisms requires all services on which they depend, to be up and running before any configuration distribution can be performed. Another drawback with NFS is to limit the database to be implemented using only flat files.

The *lcfg* [2] uses NIS to distribute configuration files that are stored in the central database. GeNUAdmin [3] propagates the configuration information using the *rsh* command, and retrieves data and programs using the NFS. Its database is centrally implemented with directories and files. According to the paper, it seems that *Gutinteg* [5] uses the NFS to propagate the configuration files and software packages, which are stored in directories and files in a central server.

Another solution is to design and implement a distribution protocol using RPC. One clear advantage of this approach is that the configuration distribution system only needs the RPC system to be running. Therefore, it can be used to configure every service that is started up after the RPC system in the boot process. Besides, a simple and clear design can lead to a safe and robust protocol because it uses a standard to represent external data and does not need privileged access on the remote hosts.

AUTOLOAD [10] and *Aurora* [7] use RPC based protocols. The former implements the mechanisms to propagate configuration and software package information from a central database that is implemented using flat files.

For the reasons cited above, the design and implementation of an RPC based protocol is the most interesting approach, despite its inherent complexity.

This is also the approach used to implement the system presented in this article.

Almost all the systems presented above make use of a central database structured as a hierarchical set of files and directories. The system that is proposed in this article does not impose any database organization or architecture. The database can be implemented using a relation/object-oriented DBMS or flat files. Moreover, it supports several architectures: central, distributed or replicated.

When a configuration is modified in any host, the systems cited above use either a pull or a push mechanism, never both. CDS supports both pull and push mechanisms to propagate configurations.

## The CDS Architecture

CDS allows host and service configuration information to be consistently defined, stored in a database and propagated to the corresponding hosts on a heterogeneous network. The propagation of the configuration information is controlled by a uniform, simple and efficient protocol, designed especially for the system.

Using automated tools to define the service configuration, the network administrator initializes the configuration process of a given service. After this initialization, the protocol reads the configuration information from the database and transfers it to the hosts, where specified operations are performed. The database provides the following features to the system:

- **Consistency and uniformity of the configurations**: achieved through the verification of configuration information stored in the database, according to pre-defined rules also coded in the system.
- **Automatic host and service reconfiguration**: enforced by the configuration stored in the database that stays available even if the host is down (provided the database server is running).
- **Large-scale network scalability**: accomplished by the automatic propagation of the configuration information.
- **Easy inclusion of new services and platforms**: supported by the capability of defining meta-configurations in the database.

Figure 1 shows a schematic view of the CDS architecture and its components. Each component of the architecture is discussed below:

- **Administration ToolBox (ToolBox)**: a set of integrated tools used by the network administrator to define and configure hosts and services in the database.
- **Configuration Database (Database)**: a data repository that holds information about the configurations and their distribution. For each configuration, the Database holds information about the target hosts and the actions that must be executed in each host after propagation. The administrator manages the Database using the automated tools presented by the ToolBox.
- **Configuration Protocol (Protocol)**: component that performs the distribution over the network of the configuration information stored on the Database to the hosts. It has two sub-components:
  - **Configuration Protocol Server (Server)**: software component responsible for reading the configuration information from the Database and propagating it to the corresponding host through the Protocol. The architecture supports several Servers in a given network to obtain high degree of performance and resilience.
  - **Configuration Protocol Client (Client)**: software component that interacts with the Server to receive the configuration information stored in the Database and performs the operations to configure the host on which it is running. Every host on the network can be managed as long as it executes this component. In fact, hosts can run both the Server and Client components if necessary.
- **Database Access Interface (Interface)**: component used by the Server to interact with the database management system (DBMS) to access the configuration information held in the Database.
- **Trap**: mechanism that allows the ToolBox to indicate the presence of modifications in the configuration of a given host. This element defines an immediate method of distributing configuration.

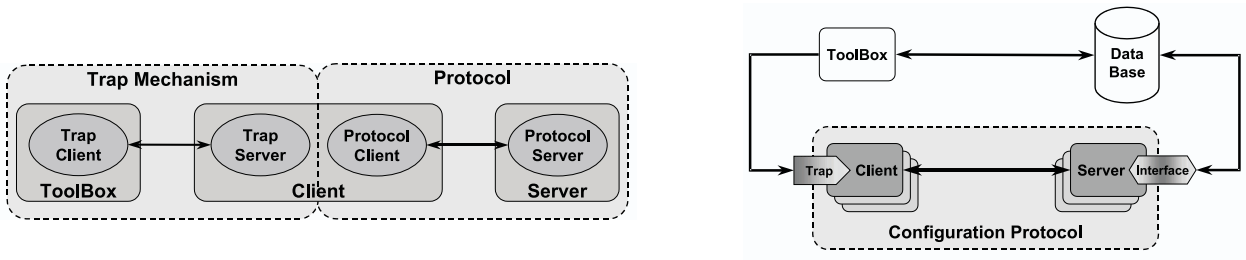


**Figure 1**: System's Architecture.

The system's architecture provides the independence between its components, allowing the design and implementation of each component to be performed in an autonomous way. This independence is supported in the following way:

- **Protocol/Database**: the operations performed by the Interface implement the independence between the Protocol and Database. To use the Protocol with a given DBMS, only the Interface must be modified, preserving the Client and Server codes.
- **Protocol/ToolBox**: the model and structure of the configuration information stored in the Database implement the independence between the Protocol and the ToolBox.
- **ToolBox/Database**: another interface can be used to implement the independence between the ToolBox and Database. This interface has not been implemented in this version of the system.

The FLASH Project [12], where this work is developed, has activities centered in each system's component. The following section presents the Protocol's architecture, components and operations.

## The Configuration Protocol

As discussed before, the Protocol was designed using the Remote Procedure Call (RPC) paradigm. RPC allows access to remote services through a procedure-oriented interface, which is based on the client-server model. The RPC server is a program that implements a set of procedures that are called by the RPC clients. A program number and procedure numbers internally represents the program and its procedures.

To support differences in data representations among several platforms and network technologies, RPC uses the External Data Representation (XDR) standard, which provides common data representation over the network. RPC/XDR ensures portability across different hardware platforms, operating systems, network architectures and transport protocols.

### Activation Modes

The configuration propagation must be robust even in conditions of high network and CPU load. For this reason, the Protocol ought to reduce the traffic over the network and the processing load in the hosts. In addition, the Protocol must offer a mechanism for immediate configuration of services. The Protocol reduces the traffic load using messages that carry a small amount of information. Furthermore, the Protocol's operations generate negligible traffic when the host does not require modification in its configuration.

The network's resources are used more intensively only when an action requires the transference of a file stored in the Database. However, in this case, the Protocol divides the file in blocks that are transferred over independent messages. This message partitioning avoids peaks of network traffic.

To minimize CPU load on the Servers and to allow the immediate configuration of services, the Protocol has two activation modes:

- **Pull Mode**: the Client periodically activates the Protocol at time intervals defined by the administrator. In such case, the activation control is distributed among the Clients, and thus, reduces the processing load on the Servers.
- **Push Mode**: the Client activates the Protocol upon the receipt of a trap signal sent from the ToolBox. This signal forces the immediate propagation of a new configuration towards the corresponding Clients.

### Structure of the Configuration Information

The Protocol employs the concepts of tasks and actions to define the configuration structure, as shown in Figure 2. A host configuration is an ordered set of tasks, where each task is responsible for configuring a given service. A task defines an ordered set of actions, where each action performs a phase in the configuration process of a given service, and thus, must be

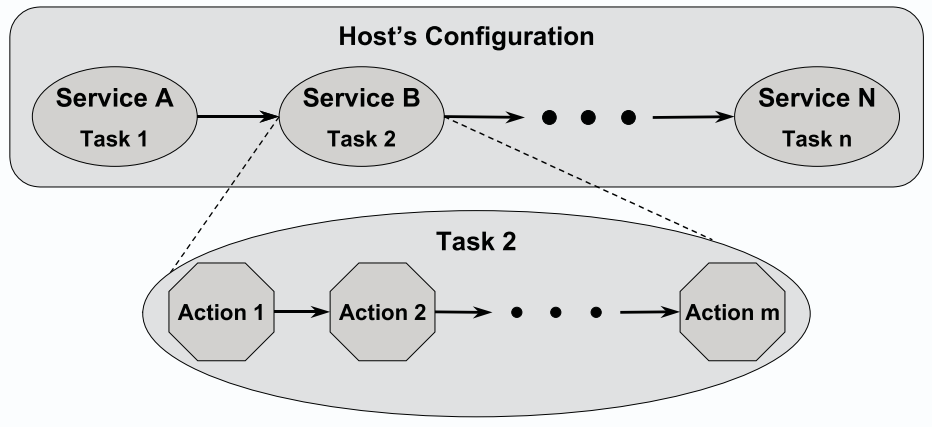


**Figure 2**: The Configuration Structure.

transferred to and sequentially executed on the Client side. Upon the execution of all tasks, the services of the Client host are properly configured.

For instance, to configure a given NFS server to export a new file system, the actions would be first to copy the modified NFS export file onto the server and then either to execute a command to export the new file system or reboot it and activate the changes.

The result of executing the actions is the installation, execution or removal of programs, scripts or configuration files on the Client. There are five types of actions implemented by the Protocol:

- ***Execute***: executes a binary program on the Client.
- ***Script***: interprets a shell script on the Client.
- ***Copy***: transfers a file to the Client.
- ***Remove***: removes a file from the Client.
- ***Reboot***: reboots the Client.

The *Reboot* action can appear in any position of the action list of a given task. When a *Reboot* action appears in the middle of an action list, the Client is rebooted and then resumes the execution from the first action that follows the *Reboot* in the list. The *Execute*, *Script*, and *Copy* actions perform their operations over a file identified in the action. The Protocol defines that this file can be stored in the following places:

- **Client's File System**: the file is directly manipulated by the action.
- **Configuration Database**: the file must be transferred from the Database to the Client, and then, manipulated by the action.

**Architecture of the Service**

The Protocol is based on the RPC's client-server programming model, where the RPC client requests the configuration information and the RPC server processes the request and transmits the result to the client. Hence, as illustrated in Figure 3, the RPC client and server are implemented on the Client and Server, respectively.

The trap mechanism is also based on the RPC model. In such case, the RPC client sends the trap signal to the RPC server, which processes the signal initializing a new Protocol iteration. On that account, as shown in Figure 3, the RPC client and server are implemented on the ToolBox and Client, respectively.

The Protocol uses the concept of stateless Server, which does not need to keep information about the Protocol's state on the Clients. This feature allows the recovery from failure on the Server to be performed in a simple way.

**The Protocol's Operations**

The Protocol is defined by a set of procedures called by the Clients and processed by the Servers. These procedures are synchronous, that is, once the procedure finishes its execution, the Client can assume that the operation has been completed and any data bound to the request are stable in the Database.

The Protocol is defined by the following procedures:

- **Request Server**: identifies the Servers currently available on the network.
- **Send Program**: stores in the Database the RPC program number used by the Client to process the trap signals.
- **Request Tasks**: retrieves the task list to be configured on the Client.
- **Request Actions**: retrieves the action list of a given task.
- **Read File**: retrieves from the Database a file associated with an action.
- **Confirm Action**: stores in the Database the successful conclusion of an action.
- **Confirm Task**: stores in the Database the successful conclusion of a task.
- **Confirm Configuration**: stores in the Database the successful configuration of the Client.
- **Send Error**: stores in the Database some information about an error identified by the Client while an action was being performed.

The Protocol represents each host, task and action using a single identifier stored in the Database. These identifiers are used as arguments for and answers from the procedures. Their representation formats are dependent of the DBMS used to implement the Database. Therefore, the Clients and Servers must manipulate the identifiers as a set of non-interpreted bytes that indicate the host, task or action manipulated by the procedures. The ToolBox generates the identifiers when a host, task or action is created in the Database. These identifiers implement a simple protection mechanism since the Server can check, for

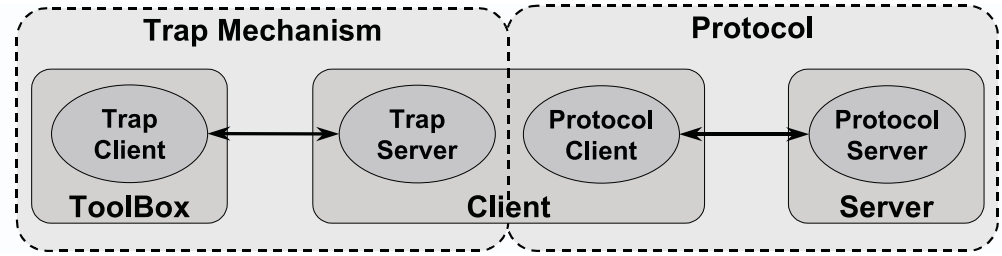


**Figure 3**: Service Architecture.

instance, if a given action belongs to the specified task or a given task belongs to the specified Client.

**The Configuration Process**

In both pull and push mode, the configuration process is implemented through the following sequence of operations on the Client side:

- **Select Server**: Client calls the *Request Server* procedure to identify Servers currently available on the network (Figure 4). This procedure must be called in broadcast mode, allowing every available Server to respond. Once the Servers have been identified, the Client selects one using an availability criterion. If no Server responds, the *Request Server* procedure will be called after waiting for a random period of time.
- **Send Trap Program Number**: Client calls the *Send Program* procedure to register in the Database the RPC program number used to process traps (Figure 5). This procedure is called only during the first activation of the Protocol.
- **Request Configuration**: Client calls the *Request Tasks* procedure to request the task list $[T_1, \ldots, T_n]$ to be configured. The Server retrieves the Client's task list from the Database and transfers it to the Client. If the Server does not find any task to be configured, it will send an empty list. In such case, the Client finishes the current interaction (Figure 6).[2]
- **Process Tasks**: for each task $T_i$, the Client sequentially requests, performs and confirms each action that composes its action list $[A_{ij}, \ldots, A_{im}]$. This is explained in detail below:
  - **Request Action List**: Client calls the *Request Actions* procedure to request the action list $[A_{i1}, \ldots, A_{im}]$ of the task $T_i$. The Server retrieves the action list of the task $T_i$ from the Database and transfers it to the Client. If the Server does not find any action to be performed, it will send an empty list. In such case, the Client must confirm the task calling the *Confirm Task* procedure.

[2] All other procedures have similar interaction model..

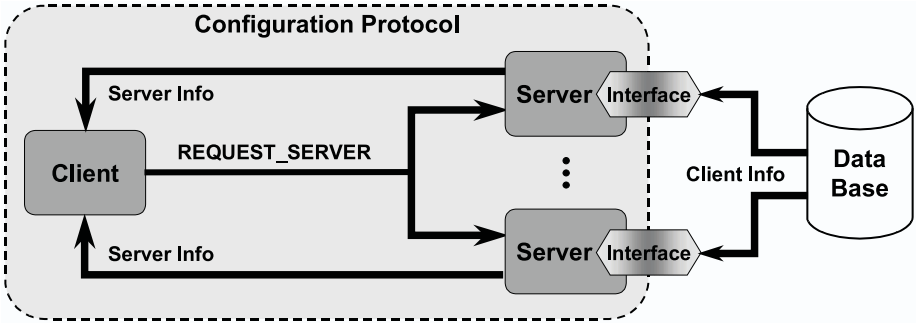


**Figure 4**: Select Server.

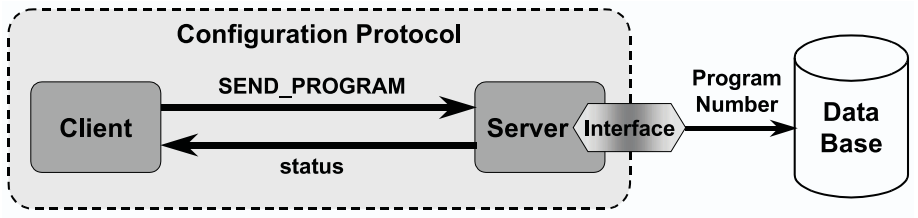


**Figure 5**: Send Program Number.

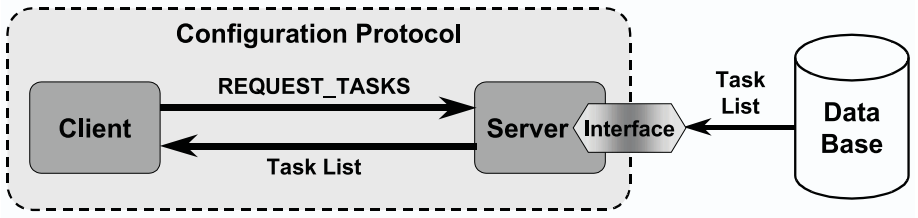


**Figure 6**: Request Configuration.

- **Process Actions**: Client sequentially performs and confirms each action $A_{ij}$ of task $T_i$:
  - **Perform Action**: Client performs the action $A_{ij}$ executing the specific functions of the action type. In *Execute*, *Script*, and *Copy* actions, if the file to be manipulated is stored in the Database, the Client will use the *Read File* procedure to read data blocks of the file.
    If the Client detects an error during this phase, it will call the *Send Error* procedure to register in the Database an information about the error. Immediately after the error is registered, the Client suspends the current iteration and disables the pull mode of the Protocol's activation. By doing that, a new iteration will be started only upon the reception of a trap. This behavior avoids the Client to be kept trying to perform an action that has an error. The administrator must correct the action and then send a trap signal to the Client.
  - **Confirm Action**: upon the successful conclusion of the action $A_{ij}$, the Client calls the *Confirm Action* procedure to confirm the action in the Database.
- **Confirm Task**: if all actions of the task $T_i$ were confirmed, then the Client calls the *Confirm Task* procedure to confirm the task in the Database.
- **Confirm Configuration**: if all tasks of the Client were confirmed, then the Client calls the *Confirm Configuration* procedure to define the conclusion of the configuration in the Database.

In case of failure of a given Server, the Client can continue the current Protocol iteration with another Server, exactly from the point where the failure has happened. To do that, the Client must identify other Servers using the *Request Server* procedure, and then, continue the interaction with the selected Server. So, when the Client receives an RPC error, it must call the *Request Server* procedure to identify another Server and continue with the Protocol iteration.

The confirmation procedures allow a given Client to reboot and continue the configuration process without redoing actions and tasks previously confirmed. In such case, when the Client calls the *Request Tasks* and *Request Actions* procedures, the Server informs only the tasks and actions that were not configured yet.

**The Trap Mechanism**

The trap mechanism allows the administrator to indicate modifications in the configuration of a given host. It is composed by a single RPC procedure. This mechanism forces the Protocol activation without waiting for the pull mode. Taking into account the Client's state when a trap is received, its processing can be performed in two ways:

- **Immediate Mode**: the Protocol is immediately activated on the Client when the trap is received during the waiting time of the pull mode.
- **Delayed Mode**: the Protocol activation is delayed in the Client when the trap is received during an iteration of the Protocol. In this case, the trap must wait the conclusion of the current iteration, and then, it activates a new iteration.

The trap mechanism has a single RPC procedure, called *Send Trap*, which is responsible for sending the signal from the ToolBox to a given Client (Figure 7).

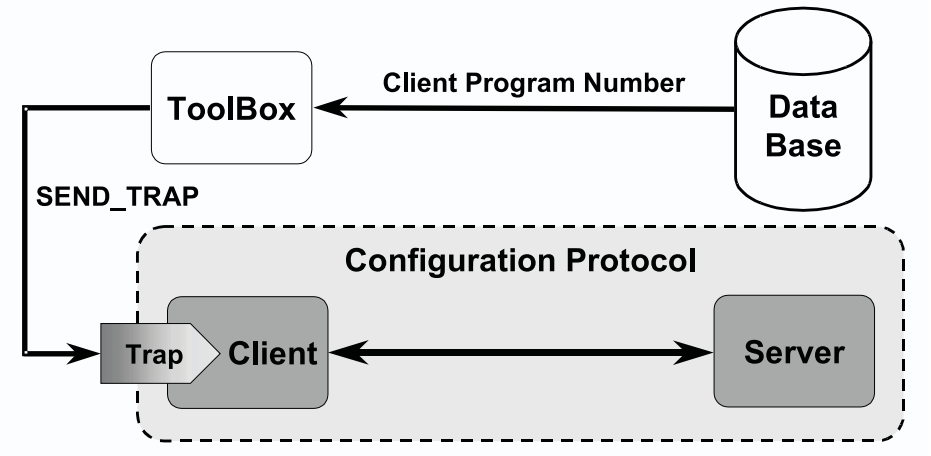


**Figure 7**: Send Trap.

Initially, the ToolBox reads the Client program number from the Database. This number was previously stored when the Client activated the first iteration of the Protocol and called the *Send Program* procedure (Figure 5). Then, the ToolBox calls the *Send Trap* procedure to notify the presence of modifications in the Client's configuration. According with the Client's state, the Protocol is immediately activated or delayed, as discussed above.

Upon the receipt of the trap signal, the Client instantly sends a status code to the ToolBox. This code indicates the current state of the Client: executing the Protocol or waiting the timer of the pull mode. The ToolBox is unblocked when it receives this status code.

**The Database Access Interface**

The Interface is composed of a set of eleven procedures, directly related to the operations supported by the Protocol. These procedures perform the data representation format conversion between the Protocol and the Database:

- **Open Connection**: opens the connection with the Database. Before accessing the Database, a Server must open a connection with the DBMS.
- **Close Connection**: closes the connection with the Database. After identifying any error, the Server must close the connection with the DBMS.
- **Request Host_Id**: retrieves the Client's identifier from the Database. The Client retrieves its identifier through the *Request Server* procedure (Figure 4).

The other eight procedures of the Interface have names and functionality that are similar to the Protocol's procedures: *Send Program*, *Request Tasks*, *Request Actions*, *Read File*, *Confirm Action*, *Confirm Task*, *Confirm Configuration*, and *Send Error*.

For example, when the Client calls the *Request Tasks* procedure of the Protocol, the Server activates the same procedure of the Interface to retrieve the information from the Database (Figure 8). In this case, the Interface converts the data format between the DBMS and Server representation, and then, the Server sends the information to the Client.

The Interface allows the Protocol to be independent of the DBMS. To use the Protocol with another DBMS, only the Interface has to be changed. The Interface allows the Server to remain connected to the Database during its execution, without opening a new connection for each request. This behavior enhances the performance of the system.

**The Implementation**

An implementation was developed to validate and evaluate the system's architecture and the Protocol. This implementation was developed in C language for Sun Solaris 2.5.1. The availability of the current implementation to others platforms depends on the portability of the C language and RPC library.

The design and implementation of the ToolBox, the data model of the Database and the interaction mechanism between ToolBox and Database are not included in this paper. However, to validate the Protocol implementation, these elements (ToolBox and Database) were developed.

The Database was implemented using the O2 object-oriented database management system [13]. The Database has a set of object classes to represent the configuration structure, pointed out previously. The Toolbox was implemented as a set of programs in O2C, a language that belongs to the O2 System. It has been used and tested to perform some tasks to configure the NFS service.

The Protocol was developed using Remote Procedure Call (RPC) [11] and its Client and Server programs were written in C. The Interface was coded in C to access the O2 database and was integrated to the Server.

The implementation has full functionality according to the specification of the Protocol. Future works must include security and access control mechanisms among Clients and Servers. Initially, we have to evaluate the encryption mechanisms provided by RPC platform, for instance, the Data Encryption Standard (DES).

The Client can be protected by defining a new kind of service that identifies the trusted Servers for a given Client. So, when the Client retrieves the list of available Servers through the *Request Server* procedure, it can check if the Servers are reliable. The Server processes any request only when the host, task and action identifiers belong to the host sending the request.

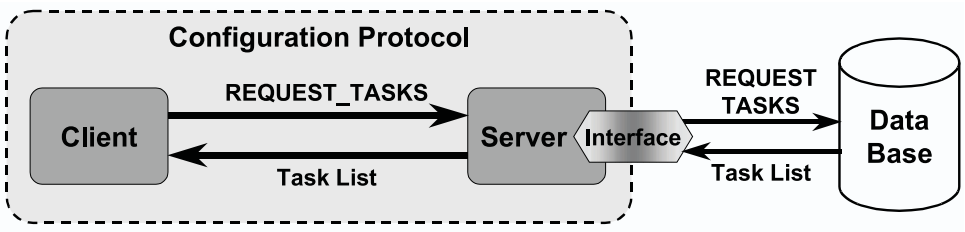


**Figure 8**: Interface's Procedures.

The broadcast mechanism used by the Protocol facilitates the initialization of the Clients and Servers, but does not allow a Client to use a Server in a different physical network. Multicast mechanism can be used to allow Servers and Clients to be in different networks.

The Server is executed as a single thread. This feature imposes some processing delays in medium-size networks. In large networks, these delays can become noticeable. To eliminate this problem, a new version of the Server can be developed using multi-threads.

### Deployment and Use of the System

To use the system, the administrator must install and activate its components appropriately. Figure 9 shows a typical installation and activation of the system.

First, the DBMS used to keep the Database must be activated in the network. In our case, the O2 System must be activated. The Clients do not access the Database directly. They retrieve their configuration information through the Servers. So, the Client hosts do not need execute any software related to the DBMS. On the Server side, when the DBMS supports access over the network, only one host executing the DBMS is needed. The current implementation needs only a single host executing the O2 System.

The next step is to choose the hosts that will act as Servers. There must be at least one Server in each network segment. To obtain higher levels of resilience, it is suggested to install more than one Server in each segment. To minimize traffic on the network and increase the performance of the system, it is suggested to install a Server and the DBMS software on the same host.

Since the Servers are installed, the administrator must install and activate the Client in each host that needs its configuration to be managed. Even the Servers have to execute the Client code if they need to be managed. This is also valid for hosts executing the DBMS software.

At this point, the system is running and the services can be configured using the ToolBox. As discussed before, different implementations of the ToolBox, with different functionality, can be constructed using the CDS as the configuration propagation and activation platform. In [9], a full-scale configuration management system is described, that uses CDS as the underlying distribution mechanism.

### Concluding Remarks

The Configuration Distribution System (CDS) presented in this article has several important features, including:

- **Simplicity**: the action types of the Protocol are simple to implement and understand, avoiding the complexity of specific commands for each service.
- **Flexibility**: the Protocol is sufficiently flexible since it supports arbitrary commands in a sophisticated way.
- **Adaptability**: the set of action types allows any service to be configured on a host.
- **Genericity**: the distribution system is generic and can be used with any other tool that support configuration planning, consistency checking and configuration file generation, as explained above. It can also be used without such system, in which case the configuration files would be manually generated.
- **Stability**: the Protocol is stable since its definition (and therefore its implementation) remains fixed even if new services or platforms need to be added to the system.
- **Performance**: the experiments have shown that the Protocol has an excellent performance, obtained by the simplicity of its operations and incremented by the dynamic distribution of the

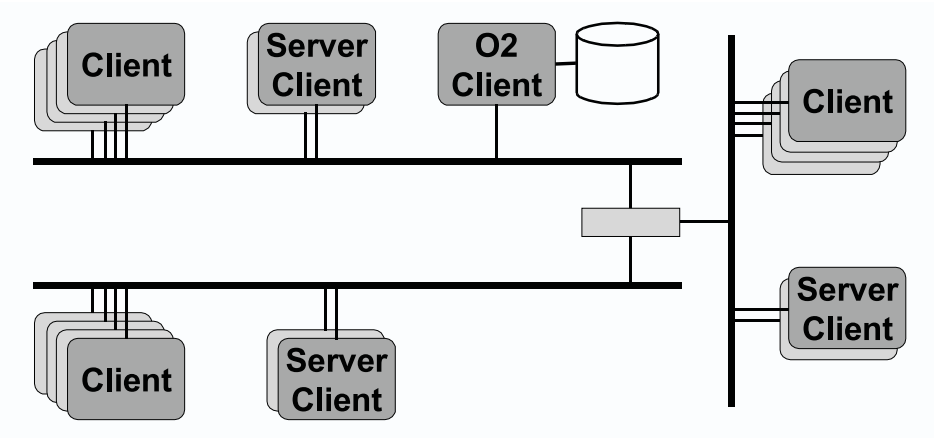


**Figure 9**: Typical Environment.

processing load on the Servers, that is implicit in the selection process of the Server.
- **Resilience**: the stateless behavior of the Protocol and the possibility of defining several Servers in a given network confer a high level of resilience to the system.
- **Robustness**: the set of facilities pointed out above makes the system robust even in high load situations.

This set of features gives to the system exceptional advantages in large-scale heterogeneous networks:

- **Service and Platform Independence**: the Protocol's operations are suitable to support the configuration of any service across different platforms. The particularities of the services and platforms were abstracted away for the Protocol and inserted in the files and commands manipulated by the actions.
- **Database Independence**: the Interface provides independence from the particular choice of implementation of the Database. To support a new kind of DBMS, only the Interface must be developed. Standard access mechanisms, for instance, ODBC or JDBC, can implement a generic interface, which operates on any DBMS supporting the mechanism.
- **Integration of Administration Tools**: the Protocol allows the integration of administration tools through the Database. In such view, different tools access and store configuration information in a common database, and, the Protocol propagates that information to the hosts, independently of how the configuration has been generated.

The facilities of the Interface can be used to design a new architecture based in multiple Databases with different structures and formats. In such architecture, the Database must store the location of the host's configuration. Prior to process a request, the Server identifies the location of the Client's configuration, and then processes the request.

When compared with existent solutions, in particular those presented in [2,3,4,5,6,7,10], the system presents a number of advantages: a) new services and platforms can be added naturally by constructing their specifications in the Database. b) consistency is greatly improved by the DBMS, instead of flat files, as in all works cited above. c) the configuration information can be used by several others system administration tools merely by accessing the Database. d) the system is, in fact, a generic framework that can be extended to support application and software package management, although to show how this can be done is out of the scope of this paper.

The implementation is stable and has been used to assist the administration of the network of the Department of Informatics, UFPE.

## Acknowledgements

The FLASH project is co-funded by the Brazilian Government agency CNPq, through the ProTeM-CC Program (Phase III) and by the Center for Advanced Studies and Systems at Recife (CESAR). Glêdson E. da Silveira receives a scholarship from CAPES.

## Availability

The system is currently packaged for distribution and can be retrieved from http://www.di.ufpe.br/~flash.

## Author Information

Glêdson E. da Silveira is a lecturer of the Department of Informatics at the Federal University of Rio Grande do Norte (DIMAp/UFRN), Brazil. He holds a M.Sc. in Computer Science from the Catholic University (PUC) from Rio de Janeiro, Brazil. Currently, he is a Ph.D. student in the Department of Informatics at the Federal University of Pernambuco (DI/UFPE), Brazil. Reach him electronically at ges@di.ufpe.br.

Fabio Q. B. da Silva is an Associated Professor of the Department of Informatics at the Federal University of Pernambuco, Brazil, where he coordinates the FLASH Project. He holds a Ph.D. in Computer Science from the University of Edinburg, Scotland. He is also the Finance Director of the Center for Advanced Studies and Systems at Recife (CESAR), a not-for-profit organization dedicated to promote Industry/University interaction (http://www.cesar.org.br). Reach him electronically at fabio@di.ufpe.br.

## References

[1] J. S. da Cunha, G. E. Silveira, F. Q. B. da Silva and J. N. de Souza. "An Object-Oriented Service Configuration Management System." *International Conference on Telecommunication (ICT-98)*, Chalkidiki, Greece, June, 1998.

[2] P. Anderson. "Towards a High-Level Machine Configuration System." *8th USENIX System Administration Conference (LISA VIII)*, San Diego, September, 1995.

[3] M. Harlander. "Central System Administration in a Heterogeneous UNIX Environment: GeNUAdmin." *8th USENIX System Administration Conference (LISA VIII),* San Diego, September, 1994.

[4] J. P. Rouillard and R. B. Martin. "Config: A Mechanism for Installing and Tracking System Configurations." *8th USENIX System Administration Conference (LISA VIII)*, San Diego, September, 1994.

[5] M. Fisk. "Automating the Administration of Heterogeneous LANs." *10th USENIX System Administration Conference (LISA X)*, Chicago, September, 1996.

[6] I. Hideyo. ‘‘OMNICONF: Making OS Upgrades and Disk Crash Recovery Easier.’’ *8th USENIX System Administration Conference (LISA VIII)*, San Diego, September, 1994.

[7] Xev Gittler, W. Moore, J. Rambhaskar. ‘‘Morgan Stanley’s Aurora System: Design a Next Generation Global Production Unix Environment.’’ *9th USENIX Systems Administration Conference (LISA IX)*, 1995.

[8] Hal Stern, *Managing NIS and NFS*, O’Reilley & Associates Inc., 1991.

[9] Fabio Q. B. da Silva, Juliana S. da Cunha, Danielle M. Franklin, Luciana S. Varejao and Rosalie B. Belian. ‘‘An NFS Configuration and Management System and its Underlying Object-Oriented Model.’’ *12th USENIX System Administration Conference (LISA XII)*, Boston, December, 1998.

[10] D. Pukatzki e J. Schumann. ‘‘AUTOLOAD: The Network Management System.’’ *6th USENIX System Administration Conference (LISA VI),* Long Beach, October, 1992.

[11] Sun Microsystems Inc. *RPC: Remote Procedure Call – Protocol Specification V2. RFC 1057*, June, 1988.

[12] FLASH Project. <http://www.di.ufpe.br/~flash>.

[13] O2 Technology Inc. *The O2 System User Reference.* November, 1995.