Process Control Daemon

SOFTWARE HIGH-LEVEL DESIGN DOCUMENT

Document Revision:	**0.3**
Document Status:	**Preliminary**
Last Updated On:	**June 1st, 2010**
Last Updated By:	**Hai Shalom**

Contributors to this document:

Revision History

Revision	Author	Date	Comments
0.1	Hai Shalom	10-03-2010	Draft
0.2	Hai Shalom	23-05-2010	Updates after preliminary presentation
0.3	Hai Shalom	01-06-2010	Final version

TABLE OF CONTENTS

1. Introduction

- The purpose of this document is to specify the software design of the Process Control Daemon, according to the requirements document.
- The purpose of the Process Control Daemon is to provide a management tool which controls the system initialization process and monitors resources and processes.

1.1. Glossary

TBD	To be defined
API	Application Program Interface
OS	Operating system
PCD	Process Control Daemon
IPC	Inter Process Communication

2. Operating System

The Process Control Daemon API and implementation are meant to run under Linux OS.

3. Process Control Daemon requirements

The Process Control Daemon requires the Linux API and IPC library.

4. Process Control Daemon context

The PCD runs on its own context as a user space application. PCD APIs are exported by a header file and a shared library.

The Process Control Daemon must be initialized before any of its users.

5. Process Control Daemon configuration

The PCD configuration is done using a textual configuration files.

The Configuration file is loaded and parsed during the PCD's startup.

5.1. Rules configuration file

The PCD Rules configuration file is compatible with the Linux shell script syntax.

The Rules configuration file contains set of rules in blocks, per each application or process which is required to be started and monitored. For each rule, the user configures a start condition, an end condition with timeout, an action to perform in case of failure, and scheduling priority of the process.

Per each rule, an application name and parameters is provided.

The same application with different parameters is considered another rule.

The rules configuration file allows inclusion of other rules file(s) in order to allow separation of rules files according to same target or component.

5.1.1. Inclusion of more rule files

The rules configuration file allows inclusion of other rules file(s) in order to allow separation of rules files according to same target or component.

5.1.2. Rule Identification

Each rule must be identified by a unique identifier. The identifier is composed of a group name, followed by an underscore and rule name (e.g. *SYSTEM_SYSLOG*).

5.1.2.1. Indexed Rule Identification

In case multiple copies of the same process handling are required, an indexed rule can be used to contain them. An indexed rule has a $ sign in the last rule name character, and must be inactive, because it is activated by the requesting application only. The PCD will start the indexed rule as many times as requested, where the first instance will have the $ sign replaced by the index.

5.1.3. Start and End conditions

The PCD can be configured to start a process only if a condition has been satisfied, and complete the rule only if an end condition has been satisfied within the defined timeout. See Condition check module section.

5.1.4. Scheduling

The PCD can be configured to setup the process scheduling (priority). The supported scheduling is either **NICE** in the range of -19 to 19, or **FIFO** in the range of 1 to 99. Note that the latter is designed to be used by real time high priority processes.

5.1.5. Timeouts

The PCD can be configured to setup timeout for the end condition. The timeout is provided in milliseconds. In case that no timeout is required (wait forever), the value -1 must be set in the correct line.

5.1.6. Failure (recovery) actions

The PCD can be configured to perform an action upon a failure. See Failure action module section.

5.1.7. Daemon flag

The user can define the process as a daemon. The PCD handles daemons as processes which can never exit. If a daemon exists, the PCD will trigger the associated failure action.

5.1.8. Active flag

The user can define if the rule is active or inactive. In case the rule is active, the PCD will enqueue the rule and will activate it as soon as its start condition will be satisfied. In case the rule is inactive, the PCD will not activate it, unless requested specifically by an application. This may be used by applications which need to be started upon a configuration or specific logic.

5.1.9. Syntax errors and run-time errors handling

In case there are syntax errors in the configuration file (i.e. misspelling, missing variables in a block), the PCD will log this error and abort. There is no error recovery from this situation, due to code space and also because the final product must have a well defined configuration file.

Run-time errors will be logged and there will be an effort to recover from them (e.g. if a process does not exist, ignore the rule and continue to the next rule).

The syntax of the PCD scripts can be checked by the host version of the pcdparser utility See offline parsing section.

5.1.10. Configuration file syntax

The syntax of the configuration file is as follows:

```
# Include a rule file
INCLUDE = filename.pcd

################### Start of a rule block ############################

# Index of the rule
RULE = GROUPNAME_RULENAME

# Condition to start rule, existence of one of the following
#
# NONE                          - No start condition, application is spawn immediately
# FILE filename                 - The existence of a file
# RULE_COMPLETED id             - Rule id completed successfully
# NETDEVICE    netdev           - The existence of a networking device
# IPC_OWNER owner               - The existence of an IPC destination point
# ENV_VAR name, value           - Value of a variable
#
START_COND = { NONE; FILE filename; PNAME pname; RULE_COMPLETED id; NETDEVICE netdev; IPC_OWNER
owner, STATUS script, status }

# Command with parameters, NONE for sync point
COMMAND = cmd parameters...

# Scheduling (priority) of the process
SCHED = { NICE value; FIFO value }

# Daemon flag - Process must not end
DAEMON = { YES, NO }

# Condition to end rule and move to next rule, wait for one of the following:
#
# NONE                  - No monitor on the result, just spawn application and continue.
# FILE filename         - The existence of a file
# EXIT status           - The application exited with status. Other statuses are considered failure
# NETDEVICE netdev      - The existence of a networking device
# IPC_OWNER owner       - The existence of an IPC destination point
# PROCESS_READY         - The process sent a READY event though PCD API.
# WAIT msecs            - Delay, ignore END_COND_TIMEOUT
#
END_COND = { NONE; FILE filename; EXIT status; NETDEVICE netdev; IPC_OWNER owner; WAIT msecs;
PROCESS_READY }

# Timeout for end condition. Fail if timeout expires. -1 if not relevant.
END_COND_TIMEOUT = msecs

# Action upon failure, do one of the following actions upon failure
# NONE - Do not take any action
# REBOOT - Reboot the system
# RESTART - Restart the rule
# EXEC_RULE id - Execute a rule
#
FAILURE_ACTION = { NONE, REBOOT, RESTART, EXEC_RULE id }

# Rule is Active or not (To be activated later by PCD API)
ACTIVE = { YES, NO }
################### End of a rule block ############################
```

5.2. Configuration Implementation

5.2.1. Data Storage

The rules data is stored by the Rules database module.

5.2.2. Parsing

The rules files will be parsed by a simple parser which will go through the configuration file and initialize the rules objects.

There is no error recovery. Each syntax error will be treated as fatal error and a log message will be displayed. The parsing process will be halted.

5.2.3. Offline Parsing

The PCD will provide an offline parser which will run on the **host** machine.
It will be used for syntax checking on the configuration file, prior to downloading it to the target.

The offline parser will reuse the parser module as is, and provide a different main function to interface with the user.

```
Usage: pcdparser [options]
Options:

-f FILE, --file=FILE            Specify PCD rules file.
-g FILE, --graph=FILE           Generate a graph file.
-d [0|1|2], --display=[0|1|2]   Items to display in graph file
                                                (Active|All|Inactive).
-o FILE, --output=FILE          Generate an output header file with rules
                                                        definitions.
-b DIR, --base-dir=DIR          Specify base directory on the host.
-v, --verbose                   Print parsed configuration.
-h, --help                      Print this message and exit.
```

5.2.3.1. Header file generation

The pcdparser will be able to generate header files which define macros with all the component's rule names. The generated names can be used by any application which wants to communicate and interact with the PCD.

5.2.3.2. Graph file generation

The pcdparser will be able to generate graph files which graphically describe each rule in the system with its dependencies. The resulted graph will be a tree, which will present the flow of the system boot sequence.

6. Process Control Daemon Core

6.1. Initialization

The PCD will be started by the system initialization script (*rcS*). The PCD will install a signal handler which will reboot the system In case the PCD is terminated for any reason.

6.1.1. Command line Parameters

```
-f FILE, --file=FILE    : Specify PCD rules file.
-p, --print             : Print parsed configuration.
-v, --verbose           : Verbose display.
-t tick, --timer-tick=t : Setup timer ticks in ms (default 20ms).
-e FILE, --errlog=FILE  : Specify error log file (in nvram)
-d, --debug             : Debug mode
-h, --help              : Print usage screen
```

6.1.1.1. File

Specifies the top level PCD script file, which contains the required rules.

6.1.1.2. Print

Print all the parsed rules on the console (not required)

6.1.1.3. Verbose

Have PCD report all events and failures

6.1.1.4. Tick

Specify the default tick. If not specified, the value is 20ms.

6.1.1.5. Error log

Specify a filename which will log all the errors in a non-volatile memory storage.
This is helpful when need to debug a crash offline.

6.1.1.6. Debug

In case a crash has occurred and a system reboot was requested as a recovery action, the PCD will not reboot the system, but leave it as is. This is helpful when need to debug a crash on the spot, where the developer can extract more information from the device.

6.1.2. Scheduling

The PCD's priority is set by default to FIFO 1.

6.2. Parser module

The PCD will have a module which will parse the rules files. The parser will go through the configuration file and initialize the rules objects using the rules database module.
There is no error recovery. Each syntax error will be treated as fatal error and a log message will be displayed. The parsing process will be halted, and the PCD will exit.

6.3. Rules database module

The Rules database module stores all the rules which are configured to the PCD by the parser module.

6.3.1. Rule object

Each object includes the following fields:

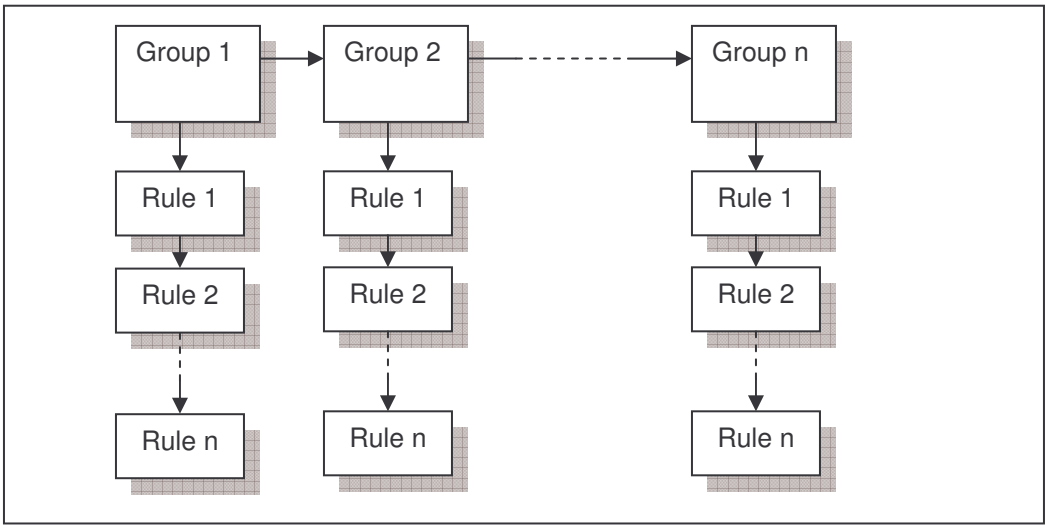
- Rule ID (as an object of component string and an integer)
- Start condition and its value
- End condition and its value
- End condition timeout
- Command (as a string)
- Parameters (as a string)
- Optional parameters
- Failure action
- Scheduling
- Rule state
- Daemon flag
- Indexed flag
- A pointer to the associated active process
- A pointer to the next object

6.3.2. Rule states

Each rule will have one of the following states:
PCD_API_RULE_IDLE: Rule is idle, never been run
PCD_API_RULE_RUNNING: Rule is running; waiting for start or end condition
PCD_API_RULE_COMPLETED_PROCESS_RUNNING: Rule completed successfully, process is running (daemon)
PCD_API_RULE_COMPLETED_PROCESS_EXITED: Rule completed successfully, process exited
PCD_API_RULE_NOT_COMPLETED: Rule failed due to timeout, failure in end condition
PCD_API_RULE_FAILED: Rule failed due to process unexpected failure

6.3.3. Rules Database storage

The rules will be stored in a sorted linked list of rule objects, for each Rule group. A rule group is defined by the Rule Prefix name (i.e. SYSTEM), in order to maximize the search performance. The objects are sorted by group identifier and name identifier.



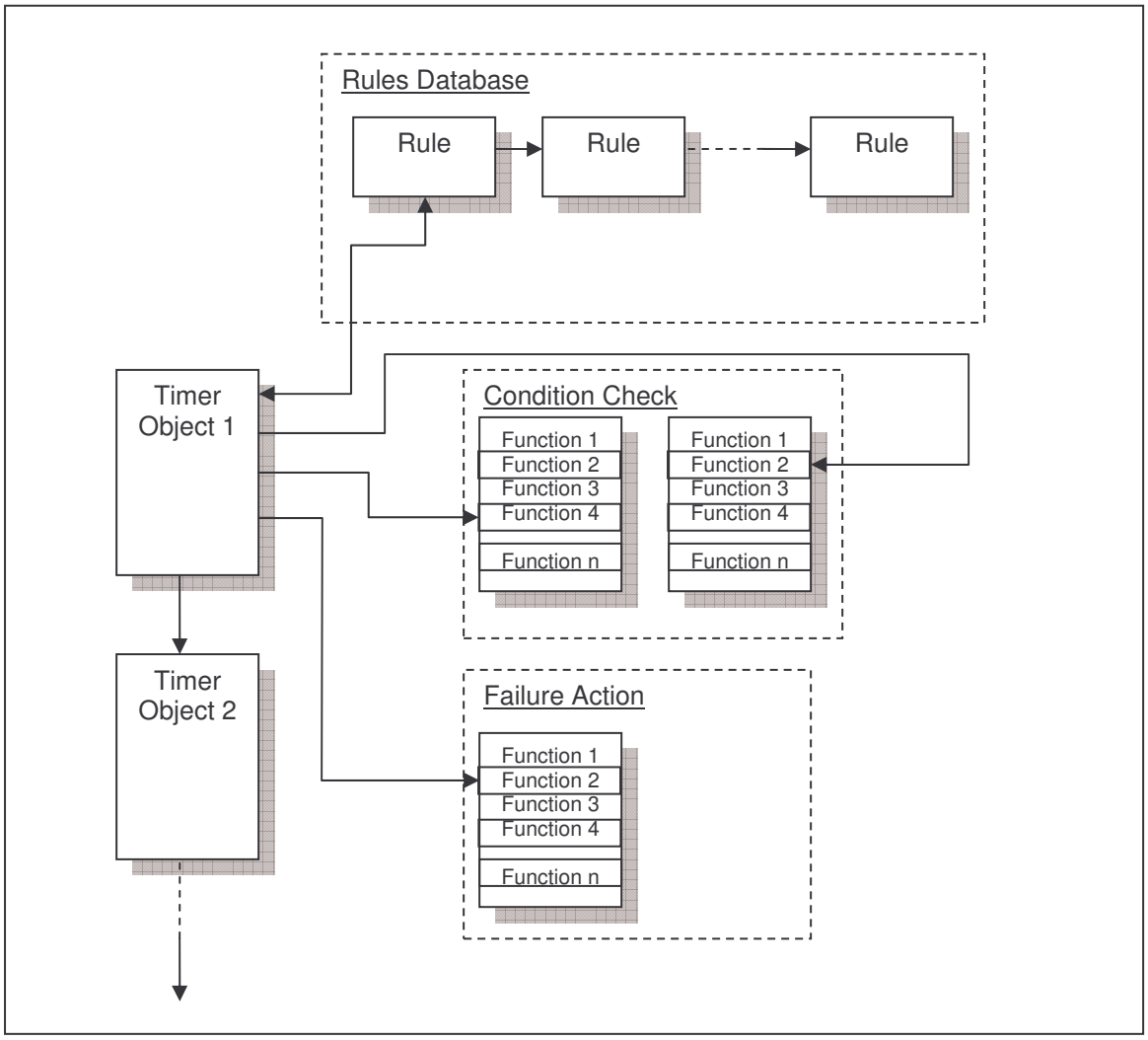
6.4. Timer service module

The PCD will have a Timer service module which will provide API to enqueue and dequeue rules to/from the timer queue. Timer ticks will be in 20ms periods, with an option to change the interval from command line. The Timer module will use the Condition check module for each rule, where queued rules will be checked by it and handled upon timeout (failure action) by the Failure action module.
Upon a successful completion of a Start condition, the process associated with the rule will be scheduled for spawning by the Process module.
Rules which were satisfied (successful completion of their End condition) will be removed from the timer queue.

6.4.1. Timer service data structure

The Timer module maintains a linked list of timer objects, which contain the active rules. The object holds the following information:

- A pointer to the associated rule
- A pointer to the rule's start condition function
- A pointer to the rule's end condition function
- A pointer to the rule's failure action function
- Time variable to hold the remaining time for the rule's timeout.



6.5. Condition Check module

The PCD will have a condition check module which will implement all the available start conditions checks and end condition checks. The module will use the native Linux API and the Proprietary TI libraries to get the required information. The condition check module will be used by the Timer module, and it will inform the Timer module if the condition check was successful or not.

The following start conditions are supported:

- *NONE* - No start condition, application is spawn immediately
- *FILE filename* - The existence of a file
- *RULE_COMPLETED id* - Rule id completed successfully
- *NET_DEVICE netdev* - The existence of a networking device
- *IPC_OWNER owner* - The existence of an IPC destination point
- *ENV_VAR name,value* - Value of a variable

The following end conditions are supported:

- *NONE* - No end condition
- *FILE filename* - The existence of a file
- *EXIT status* - The application exited with *status*. Other statuses are considered failure
- *PROCESS_READY* – The process sent a READY event though PCD API.
- *WAIT* msecs - Delay, ignore END_COND_TIMEOUT
- *NET_DEVICE netdev* - The existence of a networking device
- *IPC_OWNER owner* - The existence of an IPC destination point

6.6. Failure action module

The PCD will have a Failure action module which will implement all the supported actions which are required in case a process exited or stopped unexpectedly.

The following Failure actions are supported:

- *NONE* – Take no action
- *REBOOT* - Reboot the system
- *RESTART* - Restart the rule
- *EXEC_RULE id* - Execute a rule

6.7. Process module

The PCD will have a Process module which will start, stop, and monitor the started processes. The Process module will handle the events coming from the processes it spawns (e.g. the process was stopped, the process exited). In this case, the PCD will activate the failure action of the rule associated with the process, if required.
The Process module will provide an API to Enqueue rules which their associated processes are required to be started, as well as API to Terminate, Kill and send signals to the processes it spawned.

6.7.1. Process state

A managed process will have one of the following states:
NOTHING – Null state.
RUNME – A command to run the process is pending.
STARTING – The process is starting to run
RUNNING – The process is currently running
TERMME – A command to terminate the process is pending.
KILLME – A command to kill the process is pending.
STOPPED – The process has stopped.

6.7.2. Process exit state

A managed process will have one of the following exit states:
NOTHING – Null state (The process did not exit).
EXITED – The process has exited
SIGNALED – The process exited due to a signal
STOPPED – The process has stopped due to an error

6.7.3. Process module data structure

The Process module maintains a linked list of process objects which are associated with active (running) processes.
Each object holds the following information:

- A pointer to the associated rule
- Process state – One of the states listed above.
- Process exit state – One of the exit states listed above.
- Process ID.
- Process return code
- Signal flag – The process module has signaled the process.

6.7.4. Process module start and stop iterations

The process module will have two iteration functions which will be activated by the main process in 2 seconds ticks:

- Start iteration:
  The start iteration will spawn all pending to run objects (in RUNME state), and change the process state to RUNNING after it verified that a started process in STARTING state did not crash immediately.
- End iteration:
  The end iteration will handle pending terminate or kill requests, as well as handle all cases process exits due to all mentioned reasons.

6.7.5. Process Parameters

6.7.5.1. Static Parameters

Static parameters are specified in the Rules configuration file. Static parameters are used by default when a process is started.

6.7.5.2. Dynamic Parameters

Dynamic parameters may be specified by any process linked with the PCD API, which requires starting another process with parameters which are different than the parameters defined in the Rule configuration file.

6.7.5.3. Environment Variable Parameters

The PCD supports environment variable parameters, which are defined in the Linux environment, outside of the scope of the PCD. Such parameters can be either defined in the configuration file or using the dynamic parameters approach, and marked by a dollar-sign ($) prefix, similarly to the usage in a Linux shell.

6.8. Exception module

The PCD will have an Exception module which will handle application crashes and exceptions. The PCD will assign a socket which will be used to transfer the crash data. Each application will be able to register to default exception handlers which will gather all the crash information and send it to the dedicated socket.

6.9. Error logger module

The PCD will have an Error logger module which will log all error messages in a non-volatile storage. This will be useful for post-mortem analysis.

6.10. PCD API module

The PCD API module will create an ICC destination point for incoming messages (through ICC). The PCD API module accepts various messages, and replies with the status of the request. The PCD API module will not be running in a different thread, but it will sequentially check for messages in a non-blocking fashion.

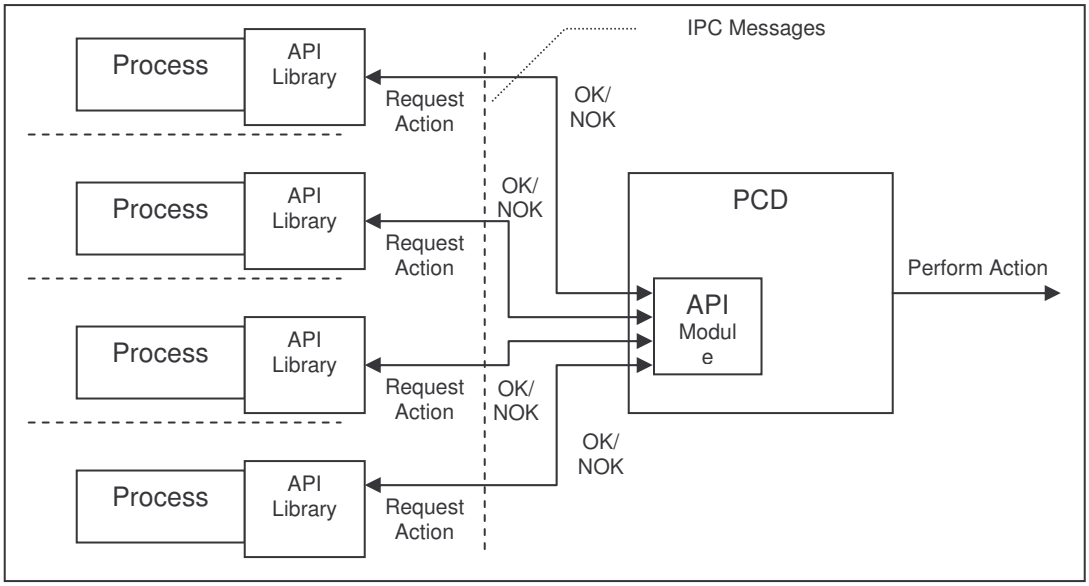
7. PCD API

The API will be available as a shared library. Each API encapsulates an IPC message to a well know PCD destination point. Each call, except a dedicated process termination API, is blocking, only for the case of immediate error, where an error return value will be returned to the caller.

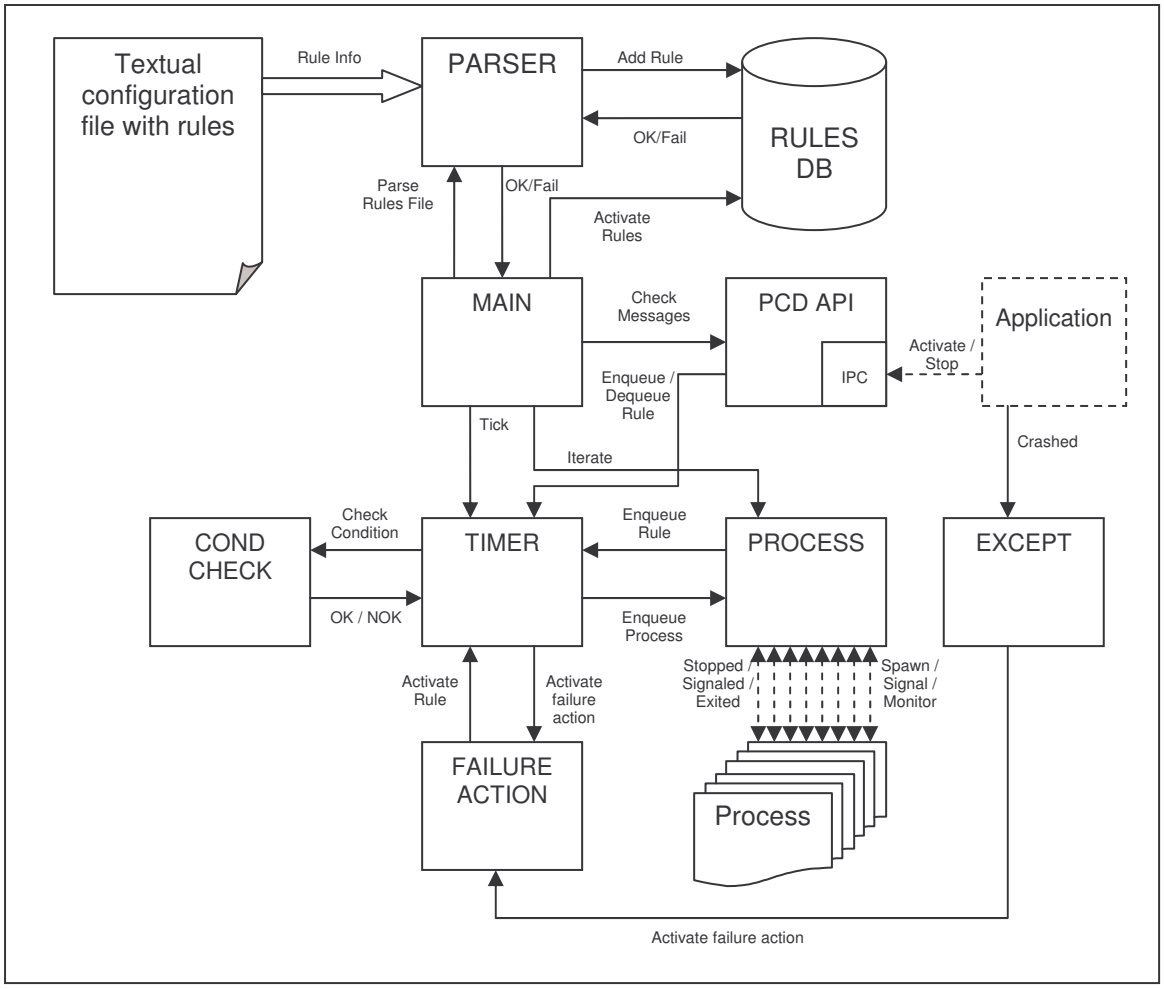
A calling process can either specify its destination point for reply, or specify -1, in case it doesn't require a reply.

The following API is supported by the PCD:

- Start a process associated with rule: Accepts Rule ID and optional dynamic parameters (May be NULL to use static parameters).
- Terminate a process associated with rule: Accepts Rule ID.
- Kill a process associated with rule: Accepts Rule ID.
- Send a PROCESS_READY event.
- Get a state of rule (for synchronization)
- Register to the PCD's default exception handlers
- Reboot the system with log



8. System description



9. Use cases

9.1. Creating a process

Creating a process is done by adding a new Rule in the Rules configuration file.
The rule could be in the state of:

- Active: The PCD will start handling the process associated with the rule immediately.
- Inactive: The PCD will start handling the process associated with the rule only upon request, using the PCD API. An inactive rule does not require CPU cycles.

9.2. Creating a Daemon process

Creating a Daemon process is done in the same way as mentioned in section 10.1. In order to specify that the process is a Daemon, there is a special DAEMON keyword, which should be marked as YES. If a rule is marked as Daemon, the PCD will consider any process exit (due to any reason, normal exit, process segmentation fault or signal) as a failure, and will trigger the configured failure/recovery action, which could be either reboot, restart the process, start another rule or do nothing.

9.3. Creating a process with a specified priority

The PCD can be configured to spawn process with a specified priority. The user can select from either to priority schemes:

- NICE: A normal process priority, which varies from 19 (lowest) to -20 (highest).
- FIFO: A high process priority, which varies from 1 (lowest) to 99 (highest). Note that 1 is higher priority than NICE -20 value.

9.4. Creating dependency between processes

Creating dependency between 2 or more processes is done using the start and end conditions of each rule. A process which requires a resource which is created by another process can specify the resource in its start condition. Another approach is that the process which creates the resource can specify the resource in its end condition and the depended process can wait for the completion of this specific rule.

9.5. Creating a process in a runtime configurable fashion

Creating a process in a runtime configurable fashion could be done by specifying an inactive rule, and activate it upon a logic which decides whether to activate it or not.

9.6. Creating numerous processes of the same executable in a runtime configurable fashion

Each copy of a process in the system requires a defined rule. In case of numerous copies of the same executable are required to run, a rule per each copy is required, because a rule is associated to a single process. The logic of how many rules to activate is done by the user.

9.7. Synchronizing processes

Process synchronization is similar to process dependency. Adding to section 10.4, each process can send a "ready event" to the PCD to specify that it is ready to accept client requests or its resources are available. It is up to the programmer to decide where and when to install this event.

9.8. Monitoring Processes Resource creation and timeout

The PCD can be configured to monitor for a process resource creation in order to determine if the rule succeeded or not. When a process's job is to create a resource (e.g. file), the resource can be specified in the rule's end condition. In case the rule contains timeout for the resource creation and the timeout expired, the PCD can be configured to trigger a failure or recovery action.

9.9. Monitoring Processes and recover from failure

The PCD monitors all the processes it spawns. Per each rule, it is specified whether the process is a daemon or not. In case the process associated with the rule is defined as a daemon, the PCD will consider any process exit (due to any reason, normal exit, process segmentation fault or signal) as a failure, and will trigger the configured failure/recovery action, which could be either reboot, restart the process, start another rule or do nothing.
In case a non-daemon process has exited abnormally (exit status not equal to 0, segmentation fault) a failure/recovery action will be activated.

10. Unit tests

The following list of tests will be done to ensure the functionality and robustness of the PCD.

10.1. Parser module test

- Test a variety of configuration files and validate that the rules objects are initialized properly
- Test faulty configuration files and validate that an error message is displayed.

10.2. Rules DB module test

- Test and validate Rule Enqueue API
- Test and validate Rule Search API

10.3. Condition Check module test

- Test and validate that start conditions are checked correctly.
- Test and validate that end conditions are checked correctly.

10.4. Failure action module test

- Test and validate that the various failure action are done correctly.

10.5. Timer module test

- Test Enqueue / dequeue APIs.
- Test execution of Failure action upon timeout.

10.6. Process module test

- Test Enqueue / dequeue APIs.
- Test starting of a process with parameters
- Test killing a process
- Test terminating a process
- Test that the module monitors correctly a process and handles signals correctly
- Kill a daemon process and make sure the failure action is triggered.

10.7. API test

- Start a process
- Start a process with dynamic parameters
- Terminate a process
- Kill a process
- Send PROCESS_READY event