Kazimir's Users Manual

Philippe DENIEL

July 2, 2008

Chapter 1

Introduction

This is the documentation for the Kazimir software. Kazimir is a log stream analyzer. It looks at several logs of several different types, try to find user defined problematic or pathologic situations in these logs and eventually (based on user-defined configuration) does action to signal or correct the situation. The reason why I developed such a tool is my own job: I was working as a sysadm and it's hard to keep an eye on every log at all time to locate problems when they occur. The detection's procedire could be easily automated, so I write Kazimir to do this work instead of me. Kazimir is written in PERL for two main reasons: it is a fully portable language, and I needed an interpretor for several part of Kazimir, and regexp management was also critical; PERL has such features, and it was a good thing to use the PERL interpretor (that is fully tested and validated) instead of re-writing one from my own. There are several similar tool like Kazimir, some are very simple, some are very complicated. I made Kazimir to be a kind of compromise between all the tools that I've seen. I also added chronological pattern matching (see below), a feature that I needed and found nowhere. I also wanted a tool that I would know well enough to modify it quickly if I had a new need. Writing a new tool was far the best way to achieve this.

Chapter 2

The syntax of the configuration file

In this section, you will learn a little more about the Kazimir objects and how to define them.

2.1 A few definitions

The syntax of the Kazimir configuration file is relatively simple. Each line or group of lines is identified by a leading tag that provides the information about the kind of thing the line is supposed to define.

The basis of all the work to be done by Kazimir is the logs. In all of this document, the word 'log' is used to represented a set of line in ASCII format, not necessarily a text file on a file system.

A log can be:

- a text file on a filesystem
- the standard output of a background process
- the lines read for a TCP client socket

In fact, Kazimir is designed to manage every Unix entities that can be access through a file descriptor on which the **getline** PERL function [1] . It would be an easy thing to modify Kazimir to make it manage another kind of logs if it fits these criteria.

Once you have the logs defined, you can search for 'Pattern' in a log. A pattern is a regular expression that can occur in a log. When the regular expression is found in the log, the time of the detection (based on log characteristics) is kept

[1] see *perldoc FileHandle* for more information.

Patterns are combined to define events. An event is the combination of the realization of pattern in logs. This combination is both based on boolean and chronological bases. For example, an event can be " If this pattern occurs AND this other pattern occur" or "If one of theses patterns occur and this other one in the following 30 seconds" Once the events are defined, you have to associate them with actions, this association is called an 'Order'. An 'Action' is basically the operation that launches an external command.

2.2 About duration definitions

Severals Kazimir settings are duration or intervals of time to be used to compare two dates. For examples, in the rest of this document, you will have parameters like *SelectTime* or *EventUpdateInt* . All these values follow the same syntax in Kazimir. Basically, a value has a suffix, valid suffixes are:

Suffix	Meaning	Value
s	seconds	1s
m	minutes	60s
h	hour	3600s
d	day	86400s
w	week	7 days
M	month	30 days
y	year	365,25 days
(no suffix)	seconds	1s

That means that '300s' is equivalent to '5m' or '300'. A value of a full day can be specified as '1d' or '24h' or '86400s'

2.3 Writing the configuration file and using the tags

The Kazimir configuration file is organized in sectionis, each of them identified by a tag. Each of the 'tag identified section' will define an object that can be a log to watch, an action or an order, a pattern or an event to look for. Each of these objects have specific aspects that must be clearly defined. For doing this, each tag will be followed by a list of definition looking like $< keyword >=< value >$. The definitions are separated by semicolons. A valid configuration line will be looking like this:

```
<tag> : <keyword1> = < value1 > ; <keyword2 = value2 > ; ...
```

An example is always clearer than a long explanation, these are valid configuration lines:

```
Log   : Name = test_cmd  ; TimeFormat = NONE ; Type = CMND_OUT ; Path = date ; UpdateInt = 4
Order: Name = ordre1 ; Event = event4 ; Action = action1 ;
Action: Name = Action1 ; Path = echo 'Perl programming is not a crime !!' ;
```

The tag that can be used are:

- Kazimir:
- Include:
- Log:
- Pattern:
- Event:
- Order:
- Action:
- Trigger:
- Repetition:
- Variable:

I did not tell you about the variable within the Kazimir configuration file. For the moment, you just need to know that a variable is initiated by the tag 'Variable:' . The way to use them will be shown later in this document.

What we will do now is having a look at each of the other tags.

Comments

Every line whose first non-blank character is a '#' is considered to be a comment and is ignored. Everything that follows a '#' will be ignore. Full blank line (only spaces or tab character) are accepted but ignored, but I strongly recommend to put some in the configuration file because it makes it a lot more readable.

Continuous lines

For some Kazimir's tags, the line can become quite long (I think more especially about the 'Kazimir' tag). Splitting a long line in several shorter lines is then natural. In most UNIX's scripting languages, the backslash character is used at the end of the line to tell to the parser that is next line is to be concatenated with the current one. This syntax is also available in the Kazimir's configuration file. Each line whose last non-blank character is a backslash will continue on the next one.

Protecting a "=" character in a command

There are objets in the kazimir configuration file that are defined by commands to be executed. These commands may contain the "=" character. This character must always be protected by a backslash. Example: never use

```
Path = Mycmd | grep "="
```

But use

```
Path = Mycmd | grep "\="
```

2.3.1 The 'Kazimir' tag

The 'Kazimir' tag is used to define the characteristics of the Kazimir program itself. This tag is the only one that should be found only once in the configuration file. It is of course possible to put several Kazimir tag but only the last one in the config file will be used (because it will be the last to be parsed). Each of these characteristic is associated with a keyword . All this keywords are mandatory, but not TmpDir, PrologueCmd and EpilogueCmd.

The keyword and and significance are

- LogFile : the location of the Kazimir log
- EventDir: the path of the directory that will receive the event completion records
- OutputDir: the path of the directory for the external command output records.
- TmpDir: the path to the directory for temporary files (usually stdout and stderr from coroutines). Default value is /tmp
- LockFile: the location of the lockfile. Kazimir does nothing as long as this file exists.
- EventUpdateInt: shows the delay between two event check passes. This is a time postfixed notation (see below)
- IdleTime: an optional tag used to define the number of seconds to stay idle between two events' checks. This could be used for avoiding a load on the CPU due to Kazimir. This value should not be to large in order not to collide with the other definitions.
- SelectTime: an optional tag, for advanced use only. The Kazimir software is mostly a loop that checks for new information coming from the logs. This

is do by a call to the 'select' function [2]. There is a timeout interval to be specified for the select. The duration for the timeout is to be set here. The default is 450ms. A value of -1 will totally disable the timeout. This is pretty useful for saving CPU (Kazimir does nothing if no new informations come), BUT it prevents you from detecting the non-production of a pattern and nothing will be displayed in the Kazimir's log if nothing happens in any log (the call to select will hang until something occurs)

- PrologueCmd : an optional tag, used to specify a command to be ran at the time Kazimir starts, before proceeding into the main event loop. The command can contain variables managed by Kazimir (see the section about variables for details). This is not a mandatory field, no command will be executed if it is not specified.

- EpilogueCmd : an optional tag, used to specify a command to be ran at the time Kazimir ends, after a SIGINT or SIGTERM was received. The command can contain variables managed by Kazimir (see the section about variables for details). I strongly suggest to put "clean up" commands in this field. This is not a mandatory field, no command will be executed if it is not specified.

A valid line will look like this:

```
Kazimir: LogFile = /var/kazimir.log ; EventDir =/var/eventdir ; OutputDir = /var/outputdir  ; LockFile = /tmp/kazimir.lock ; \
EventUpdateInt = 5s
```

The IdleTime is a field that is optional and to be used very carefully. When set it makes Kazimir sleep the duration indicated just after the Event Checking pass. During this time Kazimir does nothing. This is a good way of making Kazimir consume less CPU, but it can badly interfere with Event processing and make you looses event occurrences if set too big

2.3.2 The 'Include' tag

The 'Include' tag is simply used for including another configuration file the current configuration file. By working this way, it is possible to avoid huge configuration files (more difficult to handle) by splitting them to smaller parts.

Only one keyword is used for this tag (it is a mandatory keyword):

- Path : the path of the location of the include file to use

Example:

```
Include: Path = /etc/kazimir.conf.include
```

[2] this function from the PERL standard packages does the same as the function which the same name in C

2.3.3 The 'Log' tag

The 'Log' tag is used to defined a log. As I said before, a log is not just an ASCII file, it is a set of line, so log managed by Kazimir can be of several types. As all Kazimir objects, a log as a name that must be used to identify the log object in the config file. The keyword for 'Log' are:

- Name : The name of the log object. Mandatory
- Type : the type of the log. Mandatory. It can have the following values:
  - ASCII : the log is a regular ASCII file
  - CMND_OUT : the log is the stderr and stdout of an external command. An additional keyword UpdateInt exists. If the external command ends, it is relaunched after UpdateInt seconds.
  - COROUTINE: the coroutine is very similar to the CMND_OUT, the only difference is that a coroutine is never re-spawned.
  - TCPCLIENT: the log is what we read from a TCP client socket
- Path : the path of the log. Mandatory. This is not necessary a pathname, it could be a network address or a command line. For each type, the syntax will differ:
  - with a ASCII log, 'Path' should be a regular path, for example /var/adm/messages
  - with a COROUTINE or CMND_OUT, 'Path' is a command line, for example 'binary_log_interpreter.pl /var/my_log.binary'
  - with a TCPCLIENT, 'Path' as the shape $< portorservice > @ < host >$, for example 'daytime@localhost', '13@127.0.0.1', '13@localhost' or 'daytime@127.0.0.1'
- UpdateInt : the time to wait between two run of a log defined by a command line. This is a postfixed time value (see below). Optional, default value is 5 minutes.
- TimeFormat: the format of the time representation in the log. This keyword is a little more touchy, so I dedicated a subsection to it. Optional, default value is NONE.

The 'TimeFormat' keyword

A log is usually organized as an ordinated set of records. The date of the information recorded in a line is often explicitly provided in the line (like /var/log/messages). When a pattern is found in a log, Kazimir has to know the 'time of the pattern', which is basically the date of the record that fits the regexp associated with the

pattern. These "pattern's date" will be used in the event combination as we'll see in the next section. If the log is showing a date in each of its lines, then Kazimir should be able to interpret it from the processed line itself, but for this, it is necessary to tell to Kazimir where the date is located in the line, and how the date is displayed. The syntax for this definition is specific to Kazimir, it is a kind of regexp with additional signs that shows what part of the date is where. All the signs are '%< *character* >' and are very close to the syntax used by the 'strftime' standard C library's function.

For example, let's have a look at /var/log/messages: a line looks like this

```
Feb 20 02:35:43 gandalf unix: devinfo0 is /pseudo/devinfo@0
```

More generally a syslog line on my workstation will be : the date, the name of my workstation right after and a message. A date like the ones you can find in /var/log/messages are created via C function strftime with a format argument like "%b %e %H:%M:%S" Let's imagine that I want to define a TimeFormat for making /var/log/messages processed by Kazimir, something like

```
^%b %e %H:%M:%S trekking
```

will fit my needs.

The time conversion signs that you can use are shown in the following list. I still think that a example is a good way of explaining something, so I will use the date 'Thursday July 5, 2001 16:45:08' as basis and show for every conversion sign the correspondence with this date.

- %a : locale's abbreviated weekday name — **Thu**
- %A : locale's full weekday name — **Thursday**
- %f : same as %a but in french [3] — **Jeu**
- %F : same as %A but in french — **Jeudi**
- %b : locale's abbreviated month name — **Jul**
- %B : locale's full month name — **July**
- %g : same as %b but in french — **Jui**
- %G : same as %B but in french – **Juillet**
- %m : month number [1,12]; single digits are preceded by 0 — **07**

[3] In fact, I am french and I manipulate many logs with 'french' time format. Later, this sign could be use for another local langage date specification, for example with a switch based on an environment variable

- %d : day of month [1,31]; single digits are preceded by 0 — **05**
- %j : day number of year [1,366]; single digits are preceded by 0 — **186**
- %e : day of month [1,31]; single digits are preceded by a space — **" 5"**
- %q : daylight saving, can take value 0,H,h,w in winter (inactive daylight savings) and 1,e,E,S,s in summer (active daylight savings) – **"W"**
- %y : year within century [00,99] — **01**
- %Y : year, including the century — **2001**
- %H : hour (24-hour clock) [0,23]; single digits are preceded by 0 — **16**
- %I : hour (12-hour clock) [1,12]; single digits are preceded by a 0 — **04**
- %p : 12 hour period, AM or PM — **PM**
- %k : hour (24-hour clock) [0,23]; single digits are preceded by a blank — **16**
- %S : seconds on one or two digits. Leading 0 can be used or not — **08**
- %M : minutes. Leading 0 is permitted but not required — **45**
- %E : Epoch time. The number of seconds since 1/1/1970 (usual Unix's date format)

There is predefined TimeFormat: the 'NONE' TimeFormat. If TimeFormat as the value 'NONE', then it means that no time is shown within the line taken from the log. In this case, the time when the line was seen by Kazimir is used (aka the current time). If no time format is specified, NONE is the default value for TimeFormat.

Last but not least: The TimeFormat is very similar to a regexp, so take care of additional space characters you would leave in it. The TimeFormat will be defined by something like *TimeFormat =%b %e %H:%M:%S;* , but remember that the regexp begins IMMEDIATLY after the '=' sign and ends right before the semicolon or the end of the line. Be careful not to put *TimeFormat = %b %e %H:%M:%S;* or *TimeFormat =%b %e %H:%M:%S ;* instead of *TimeFormat =%b %e %H:%M:%S;*. This could make the TimeFormat impossible to find in the line.

Time coherency within the log is also checked. Imagine that you have a log that result in the conversion of some information received through UDP. If you are a little familiar with UDP, you know that messages are not guaranteed to be received in the order they were sent. So you can have a line whose date is older than the date of the line just written before. Kazimir will do this check:

if a pattern is seen in the log and if the date computed for the pattern is older that the date of the previous occurrence of this pattern, then the previous (and earlier !!) date is kept and a warning is printed is Kazimir's log. What happens if the TimeFormat is NONE in this specific case: nothing because the NONE TimeFormat will time-stamp every line with the time when it was seen, so they can't be time incoherency, but only in appearance because in fact you are not warned of a potentially pathologic situation. This is one of the reason why I strongly recommend not to use the NONE TimeFormat.

Here are valid Kazimir configuration line for some logs:

```
Log: Name = syslog ; Type = ASCII ; Path = /var/adm/messages ; TimeFormat=%b %e %H:%M:%S;
Log: Name = cmd_respawned  ; TimeFormat = NONE ; Type = CMND_OUT ; Path = /bin/my_command ; UpdateInt = 40s
Log: Name = cmd_not_respawned  ; TimeFormat = NONE ; Type = COROUTINE ; Path = /bin/my_bg_diagnostics ;
Log: Name = remote_log ; Type = TCPCLIENT ; Path = logport@loghost ; TimeFormat=%m/%d/%Y %H:%M:%S;
```

The 'kazimir.debug.TimeFormat' tool

Writing a correct TimeFormat is, in my opinion, one of the most difficult thing when writing a Kazimir configuration file. It is pretty easy to use a bad format and there is a need for a debugging tool. So I wrote a tool called *kazimir.debug.TimeFormat*: it's a command line script with two arguments: the first is a TimeFormat to test, and the second is a sample of a line from the log where the TimeFormat is supposed to work on. This tools displays several pieces of information like: the regexp computed from the TimeFormat, and several information got from the parsing process. It says at the end if the format is correct, the epoch time read in 'integer' format and in 'localtime' format too. So checking the coherency is possible. Again, I think that an example is better than a long discussion: I have to cop with file /var/adm/messages, but I don't know which one of these two time format I should be using between '%b %e %H:%M:%S' and '%b %d %H:%M:%S' .

I will use the two following commands (based on a sample from my workstation) :

```
kazimir.debug.TimeFormat '%b %e %H:%M:%S'
\  'Mar  1 13:43:35 trekking.bruyeres.s last message repeated 4 times'
kazimir.debug.TimeFormat '%b  %d %H:%M:%S'
\ 'Mar  1 13:43:35 trekking.bruyeres.s last message repeated 4 times'
```

Here, I see that both seems correct with this line, but try now this

```
kazimir.debug.TimeFormat '%b %e %H:%M:%S'
\  'Mar 11 13:43:35 trekking.bruyeres.s last message repeated 4 times'
kazimir.debug.TimeFormat '%b  %d %H:%M:%S'
\  'Mar 11 13:43:35 trekking.bruyeres.s last message repeated 4 times'
```

We see that '%b %e %H:%M:%S' is ok, but '%b %d %H:%M:%S' is erroneous (day of month is displayed with a leading space for single digit). With this

example, I would like to bring your attention on time element displayed with a leading space on a leading zero for single digit values. It was one of my most frequent mistakes when I wrote Kazimir's configuration file for my machines.

2.3.4 The 'Pattern' tag

The 'Pattern' tag will defined a pattern object. A pattern, as every Kazimir objects has a name to identify it, and is also defined by the name of the log where it should occur and the regexp that defined the pattern. All the keywords are mandatory.

- Name : the name of the pattern
- RegExp : the regular expression that defines the pattern
- Log : the name of the log where the pattern can be found.

Examples:

```
Pattern:  Name = NoError; RegExp =NoError; Log = my_log
Pattern:  Name = CriticalOrMajor ; RegExp =CRIT|MAJOR ; Log = my_other_log
```

2.3.5 The 'Event' tag

The 'Event' tag is the most complicated tag to use. It defines an event as a boolean and chronological association of realized patterns. Events are check only during 'Event Check passes'. The delay between two of these passes is defined by the *EventUpdateInt* keyword of the 'Kazimir' tag. An event is generally associated with a 'time window'. This means that only the patterns realized within the time window will be considered. The 'time window' is a duration value (a postfixed time value). As an example, let's consider that I set up an event with a 10 minutes time window, so I'll considered only the event between the current time and the current time minus 10 minutes. The patterns that were realized before are simply forgotten. If no window time is defined, then all the pattern detected form the start of Kazimir will be considered. Be very careful while doing this because you can makes a configuration file that detects always the same old error that happened so many time ago that it should not be considered. The 'Event' tag have two keyword:

- Name: The name of the event. Mandatory
- Window: The window time. Not mandatory, the default is -1 (looks pattern form the beginning). I strongly recommend to use a window time (see remark above)

But this is not enough for defining an event, you should also describe the 'event combo' this is a set of boolean pattern association. The combo begin with a line with *Begin* and ends between with a line with *End*. Each line between this two statement are combo lines. A combo line is a boolean composition of pattern realization. Imagine I have to look for pattern named **pat1**, **pat2** , **pat3**. I can then define combo line like

```
I want the line to be realized if pat1 OR pat2 is ok:

pat1 || pat2

I want pat2 AND a realization of pat3 or pat1:

pat2 && ( pat3 || pat1 )

I want a realization of pat1 with NO realization of pat2

pat1 && !pat2
```

Each combo line that is realized has a realization time. For the moment this is the higher value within the pattern realized involved in the combo line [4] . This realization time is used for chronological association. At this point, your question should be : But how to do a chronological association ? This is very simply, to say 'this should happen after this' just write two combo line, and put the line in the correct time order. For example, I want to check for pat2 or pat3 happening after pat1: I just need to write:

```
pat1
pat2 || pat3
```

But chronological association can be a little smarter by introducing the idea of 'delay' between the combo lines. For example I can look for 'pat 2 or pat3' with the next 2 minutes after pat1 was realized for the last time. You can do this by adding the additional (and optional) keyword 'Delay' at the end of a combo line (a semicolon is necessary for the parser to know when the combo ends). The value after the 'Delay' keyword is a postfixed time value. For example, for the configuration I described a few lines above, I can write:

```
pat1
pat2 || pat3 ; Delay = 2m
```

[4]This is not a very smart algorithm. It will fit many cases, but is not very precise when complex boolean expression are used. If anybody has a smarter algorithm to submit please contact me.

By doing this I said that the second line must happen in the 2 minutes that follows the realization of the first line. Of course the 'Delay' keyword has no meaning if used in the first line (there is no previous line with a time realized to be compared with). At the end of the document, you'll see an example section with different configuration files.

2.3.6 The 'Action' tag

The 'Action' tag defines a kind of alias for a command to launch when a event is realized. All the keyword for its definition are mandatory

- Name: The name of the action
- Path: The command line (with all its arguments) to be executed when using the action.

2.3.7 The 'Order' tag

The 'Order' tag is really easy to use: it just defines a relation between an event and an action. If the event occurs and such an order is defined, then the action is realized, and the related binary will be launched. The keyword for the 'Order' definition are all mandatory.

- Name: The name of the order
- Event: The name of the event that we want to be associated with an action
- Action: The name of the action to be associated with the event

2.3.8 The 'Trigger' tag

The 'Trigger' defines a trigger. A trigger is a kind of 'meta-event'. It is an event that occurs when an event that was realized at the last event check is now not realized. A trigger is then realized when a given event triggers its state from 'activated' to 'not activated'. A trigger can be associated with an order through the 'Order' tag. You can't define a trigger based on another trigger. The keyword for the 'Trigger' are

- Name: The name of the trigger
- Event: The associated event

2.3.9 The 'Repetition' tag

The 'Repetition' is another 'meta-event'. A repetition is always associated with another event, it is realized when a given number of realization of a given event occurred without being triggered off. The keywords associated with the 'Repetition' tag are:

- Name: The name of the repetition
- Event: The event correlated with the repetition
- Number: The consecutive number of occurrences of the correlated event to be found for activating the repetition

It is possible to defined a 'Trigger' for a repetition (to detect that a repetition triggered off). It is not possible to define a 'Repetition' for a repetition.

2.3.10 A few conventions

The time postfixed value

Many Kazimir objects parameters are time value for a duration (Event Window, Event Delay, Update Interval...). A postfixed notation is defined to write the value under the shape *Numerical Value[Unit Suffix]* . The available suffix are:

- no suffix or **s** : second
- **m** : minute
- **h** : hour
- **j** or **d** : day [5]
- **w** : week (by week I mean 7 days, so 604800 seconds)
- **M** : month (this is not really objective, but I consider a month to be a 30 days duration. I know this is not very clean, but I guess nobody will use such values. If this is a problem, please let me know, I'll make a modification for this).
- **y** : year. I considered that a take is about 365.25 days, which makes 31557600 seconds. Let me know if this value does not fit for some reason.

[5] 'j' is the abbreviation for *jour*. Yes, I told you I was french and kazimir spoke totally french in a former version. I just translated (with more or less success...) some part in English when I choose to make it a free software

What happens if a ASCII logfile is renamed or deleted ?

For the typical case of ASCII logfile, there is something you should know. This kind of log files often have a 'switch log' mechanism, when the file becomes to large or after a given timeout the current version is switched and replaced by an empty new one. Generally the old version is archived. When managing ASCII log, what is interesting is to keep an handle on the last version of the log, not on an archive one, so kazimir has a mechanism for log switch detection based on the inode number. After every event check pass, it looks for the inode number of the ASCII log. If the result is different from the one obtain one the previous pass, or if the log could not be opened before (it was not there), then it Troyes to re-open the log file. So if the log switches, then kazimir will always keeps an handle to the latest version as long as the log name does not change.

2.4 Using the variables

This is in fact the second version of kazimir that I write. For previous version (that I used only internally within the company with which I work) had no variable and it was a problem to keep track of a given 'state' for precise problem detection. So I decided that the new version (the one you have downloaded) will have such feature. The idea behind variable is a make the kazimir configuration file a little more dynamic by putting for variable part within some of the values associated with the object's keyword. Variable can be use in the following keyword (this also mean that they **can't** be used in the other keywords): *Order::Action, Action::Cmd, Pattern::RegExp, Event::Window, Event::Delay, Log::TimeFormat, Log::UpdateInt, Kazimir::OutputDir, Kazimir::EventDir and within a line in the event combo definition* Each time one a this value has to be used, it is first evaluated to change the variable into its value. Variable can be modified when:

- a pattern is found
- an event is realized
- an order is used
- an action is used

2.4.1 The 'Variable' tag

The 'Variable' tag is used to define a variable with a default value. It is not mandatory to use this tag to define a variable before using it, in fact if the variable was not defined with a 'Variable' tag, then its value will be the PERL default for variable as long as a different value was not put in it. Use the 'Variable' tag to have variables with an initial value that you fully want to control. This tags require two keyword that are both mandatory:

- Name: The name of the variable
- Default: The default value for the variable

2.4.2 The variable syntax

Every variable (previously defined of not) can be use in the configuration file with a syntax looking like **$(varname)**. For example, I defined a variable 'myvar' with initial value 10 by a 'Variable' tag like this:

```
Variable: Name = myvar ; Default = 10 ;
```

I can access it (for both read and write) by using **$(myvar)**.

operators to be used with variables

Operators to be used on variable are more or less the less than in PERL, except that the '=' sign is replaced by $' < -'$. But the following operator will change the value of the variable on which they are used .

Operator	PERL equivalent	what it does
$< -$	$=$	affectation
$+ < -$	$+=$	increment
$- < -$	-=	decrememt
$* < -$	*=	multiplication
$/ < -$	/=	division
$\& < -$	&=	AND operation
$< -\tilde{}$	$=\tilde{}$	regexp operation
$. < -$	.=	strings concatenation

And more generally, every operator in PERL with '=' in it can be use by just replacing the '=' character by $< -$. When operating with variables for anything but affectation, every classical operator can be used (+, *, - , / , ...) .

The internal variables

The following variable are managed by kazimir internally. They can be used by the user, but remember that kazimir will use these variables to make information available. They can be read, but not be written by the user (kazimir will immediately overwrite them).

- $(EPOCH) : The current epoch time
- $(SECOND): The seconds of the current time
- $(MINUTE): The minutes of the current time
- $(HOUR) : The hour of the current time

- $(DAY) : The day within the month for the current time
- $(MONTH) : The month of the current time
- $(YEAR) : The year of the current time
- $(WEEKDAY) : The day within the week for the current time (1= Sunday, 7= Saturday)
- $(YEARDAY) : The number of the day within the year
- $(ISDST) : Information about local time for the current epoch time
- $(LAST_ACTION) : The last action that was used
- $(LAST_ORDER) : The last order that was used
- $(LAST_EVENT) : The last event that was used
- $(LAST_TRIGGER) : The last trigger that was used
- $(LAST_PATTERN) : The last pattern to be found in any log objects.
- $(LAST_LINE) : The last line to be found in all the log objects.
- $(LAST_ACTIVE_LINE) : The last line form the log objects where a pattern was found.
- $(LAST_EVENT_CHECK) : The epoch time of the last 'Event Check' pass.
- $(FICH_EVENT) : The last event file updated by kazimir (useful mostly in 'Action' definitions).
- $(KAZIMIR_START_TIME) : The epoch time when kazimir started

Caller's environment variables

In some case, it would be nice to get the environment variable set in the process that launched kazimir. I mean the variables defined in the calling shell by **setenv VAR VAL** in csh or **export VAR=VAL** in ksh. Kazimir defines an implicit hash table called $(ENV[]) which contains all the caller's environment. For example, $(ENV[PATH]) contains the value of the shell's path. Using this hash allow closer interaction between kazimir and its caller.

2.4.3 The 'Let' statement. Operation available on variables

Each time a event, order or action is matched or a pattern is found, it is possible to modify the value of the variables. This operation is always done after the operation implicated by the use of the object are done. For example, if an event is realized, the operation on the variable is made just after the verification of all the combo lines. To associate the realization of an object with an operation on variable, an additional keyword was introduced: the 'Let' keyword. The argument of this keyword is a string enclosed with '{' and '}' and using a regular, semicolon separated variable syntax. Again, I think that a few short examples are better than a long explanation:

I want to put the value 3 to $(myvar1) :

```
Let = { $(myvar) <- 3 ; }
```

I want to put the value 3 to $(myvar1) and to put 4 to $(my_other_var) :

```
Let = { $(myvar) <- 3 , $my_other_var <-4 ; }
```

I want to put $(var) plus $(var2) in $(my_var):

```
Let = { $(my_var) <- $(var1) + $(var2) ;}
```

I want every numerical character to disappear from variable $(my_var):

```
  Let = { $(myvar) <-~ s/(\d+)//g ; }
```

I want to increment a counter called $(cpt):

```
Let = { $(cpt) +<- 1 ;}
```

As I said before, the 'Let' keyword is just to be added to a 'Order', 'Action', 'Event' or 'Pattern' definition, for example:

```
Pattern: Name = MyPattern ; Log = MyLog ; \
RegExp=Something; Let = { $(my_var) <- 0 ;} ;
```

You can use let to modify several variables in the same 'Let' statement, but you must used the comma "," as a separator to each operation. For example:

```
Pattern: Name = MyPattern ; Log = MyLog ; \
RegExp=Something; \
Let = { $(a) <- 2 , $(b) +<- 3 , $(c) <- $(d) + 4 ;} ;
```

Using a 'Let' statement in the Kazimir tag parameters

It is possible to use a 'Let' keyword with the 'Kazimir:' definition. In the case, the variable operation in the Let statement will be done at the end of every event check This is an example:

```
Kazimir: LogFile = /dev/tty ; EventDir =/tmp/eventdir ; \
 OutputDir = /tmp/outputdir  ; LockFile = /tmp/kazimir.lock ;\
 EventUpdateInt = 5s ; IdleTime= 1 ; Let = { $(evcheckcounter) +<- 1 }

Variable: Name = evcheckcounter ; Default = 0
```

In this example, we can see that the variable named 'evcheckcounter' is a counter of the number of events checks, because it is initialized to 0 and incremented by one at the event check by using a 'Let' in the 'Kazimir' tag.

2.4.4 How can I use variables ?

Kazimir is a state machine. Variable are a way to keep track of a state at a given moment and then reuse it when appropriated. I think that variables can be used for making generic action or order. Imagine we have an action that send a mail to the administrator when an event occurred, a variable containing the result of the mail can be use as argument. By using this variable it would be possible not to write a specific action for every case. Variable are also a way to react after the modification of a situation: Imagine that you know that a bad situation will produce a first regexp in general, but another regexp when a given event is detected. By using a variable in the pattern regexp definition you can dynamically change the regexp form regexp1 to regexp2 when the expected event occurred. More generically, variables can introduce more flexibility to the product, by making the configuration file more dynamic, and look a little more like a kind of 'event checking' script than like a basic configuration file.

2.4.5 Using Hash-tables

Hastables are 'associative arrays of variables'. This is a set of variable to be accessed using a key. Since hashtable are widespread in PERL, it was easy to implement them into Kazimir. Using hastable is very similar to using variables only the syntax differs. When a variable looked like *$(var)*, an entry of a hashtable will look like *$(hash["key"])*. In this "key" can be a string or a variable, so it allowed to have syntax like *$(hash[$(key)])*.

Hastable entries can be initialized one by one using the 'Variable' tag. But if you have to initialize ten items then you will have to it ten times. So the following lines are a correct syntax:

```
Variable: Name = hash[ "value0" ] ; Default =  0 ;
Variable: Name = hash[ "value1" ] ; Default =  1 ;
Variable: Name = hash[ "value2" ] ; Default =  2 ;
```

For the moment, hashtables can not be used recursively in order to be keys for other hashtables. I mean that a syntax like *$(h1[$(h2["key"])])* is no valid syntax and will produce an error.

using variables within variables

Since there are hashtables and variables, it is a natural idea to think about using variables to be keys for hastables, or values from hashtables to be keys from hastables (may be recursively). In other word, it is natural to think about using these syntaxes

- $(v)
- $(h[k])
- $(h[$(v)])
- $(h1[$(h2[$(v)])])
- $($v))
- $($(v)[$(w)])
- and so on .....

Kazimir should accept any of these syntaxes, but there is one restriction: there should be no space in the syntax. This means that $(h [k]) or $(h[k]) are no valid syntax.

2.4.6 External command evaluation

In sh, it is very common to use a syntax which looks like

```
var=‘ls -l | grep mypattern‘
echo $var
```

The fact of being able to replace a command within back quotes by its output is something I wanted to have in kazimir, for making the construction of generic configuration files easier. So I added this feature in kazimir. In any field in which you can use variables, you can use the syntax ‘cmd‘ in the same way. For example in a 'Action' tag you can write

```
Action: Nom = RespawnToto ; Path = /usr/bin/totod ‘date‘;
```

In this example, this will start the *toto* service (not an actual one of course) with the current date as argument. If this does not seem clear enough, try this syntax in one of your Kazimir's configuration file. I am sure this will not stay fuzzy for long.

It is important to notice that the syntax with back quotes is authorized within 'Let' statement. An example like this one is perfectly correct:

```
Variable: Name = x ; Default = ‘date‘

Action: Nom = RespawnToto ; Path = /usr/bin/totod $(x);   Let = { $(x)  <- ‘date‘ ;
```

And by doing this is this (perfectly stupid) example, you will start the *totod* service with the date of the last re-spawn (contained in variable $(x)).

2.5 Launching Kazimir

Kazimir can be use in two modes: the test mode, and the standard mode. In test mode, it just parse the configuration file, and write the result of its parsing to the standard output. This is useful for typo or syntax error detection. The standard mode is the one to use to detect the problem. Kazimir is to be running as a background program, generally launches from the command like through a *nohup* call. It can also be run by root at machine's boot time. Kazimir has several flag to be used on the command line:

- -h : print the syntax help for the command line
- -f < *path* > : specifies where to find the configuration file. If this flag is not used, Kazimir looks for /etc/kazimir.conf
- -T : tells that kazimir has to be launched in the test mode.

Imagine that you wrote a configuration file in your home directory, you would launched kazimir like this:

```
kazimir -f ~/my_kazimir.conf
```

Test mode can be obtained by using this command line

```
kazimir -f ~/my_kazimir.conf -T
```

2.6 A few file configuration examples

A Basic configuration file

This configuration file just checks for a single pattern in /var/adm/messages. If found it echoes something in the output file. The pattern it looks for is "toto was here", this pattern is used in event called TotoEvent associated with Action1 via ordre1.

```
Kazimir: LogFile = /dev/tty ; EventDir = /tmp/eventdir ; OutputDir = /tmp/outputdir ;\
LockFile = /tmp/kazimir.lock ; EventUpdateInt = 5

Log: Name = totolog ; Type = ASCII ; Path = /var/adm/messages ; TimeFormat=%b %e %H:%M:%S;


Pattern:  Name =Toto; RegExp =toto was here; Log = totolog ;

Event : Name =TotoEvent; Window = 30 ;
Begin
        Toto
End

Order: Name = ordre1 ; Event =TotoEvent; Action = Action1 ;
Action: Name = Action1 ; Path = echo "Toto was here !!" ;
```

Now, run kazimir with this configuration file, then use *logger -p user.err 'toto was here'* and see kazimir working [6]

2.6.1 A boolean composite event

This is the same as the previous config file, when now we look for two patterns:

```
Kazimir: LogFile = /dev/tty ; EventDir = /tmp/eventdir ; OutputDir = /tmp/outputdir ;\
LockFile = /tmp/kazimir.lock ; EventUpdateInt = 5

Log: Name = syslog ; Type = ASCII ; Path = /var/adm/messages ; TimeFormat=%b %e %H:%M:%S;


Pattern:  Name =Toto; RegExp =toto was here; Log = syslog ;
Pattern:  Name =Titi; RegExp =titi was here; Log = syslog ;

Event : Name =TotoTitiEvent; Window = 30 ;
Begin
        Toto  && Titi
End

Order: Name = ordre1 ; Event =TotoTitiEvent; Action = Action1 ;
Action: Name = Action1 ; Path = echo "Toto was here, with Titi !!" ;
```

Now, do several *logger -p user.err 'toto was here'* and *logger -p user.err 'toto was here'* [7], makes sure the messages were output to /var/adm/messages and see kazimir working

A chronological composite event

I will not comment this one to much. Here, we look for Titi within 10 seconds after Toto.

```
Kazimir: LogFile = /dev/tty ; EventDir = /tmp/eventdir ; OutputDir = /tmp/outputdir ;\
LockFile = /tmp/kazimir.lock ; EventUpdateInt = 5
```

[6] when logging messages with **logger** remember that syslog will not print the same message several time in its log but something like *last message repeated* $< n >$ *times*

[7] again, keep in mind the 'last messages repeated....' message in syslog

```
Log: Name = syslog ; Type = ASCII ; Path = /var/adm/messages ; TimeFormat=%b %e %H:%M:%S;


Pattern:  Name =Toto; RegExp =toto was here; Log = syslog ;
Pattern:  Name =Titi; RegExp =titi was here; Log = syslog ;

Event : Name =TotoTitiEvent; Window = 30s;
Begin
       Toto
Titi ; Delay = 10s
End

Order: Name = ordre1 ; Event =TotoTitiEvent; Action = Action1 ;
Action: Name = Action1 ; Path = echo "Toto was here, with Titi !!" ;
```

Re-spawning a dead process

This is an actual sample. I would say, that contrarily to the other examples that were designed for showing the way Kazimir's works, this one is an event I really use on my machines. The context is as follows: I have a process (in my example, the binary is called 'toto'[8] and it should be restarted when it crashed. Its path is /usr/local/bin/toto.

What the following configuration file does is making kazimir use 'ps -edf — grep toto — grep -v grep' periodically. If a process named toto exists, I will see it that way, if no such process exists, when the output will be totally blank. A pattern is defined to keep track on the last time the process 'toto' was seen in the output of the 'ps' command. It is then use to defined an event, but on a negative way: event is realized when the pattern could not be found in the event window time, which means the process was not up in this interval. An action is correlated to this action, it re-spawns the process, making that way the pattern be realized and. On this example, I use explicitly 'NONE' as TimeFormat. I have said in this document to avoid using this format, but here it is pretty well adapted there, because 'ps' provides only an 'instant view' on the existing process, but nothing about the past processes, no there is no danger to identify a past information as up to date.

```
Kazimir: LogFile = /dev/tty ; EventDir = /tmp/eventdir ; OutputDir = /tmp/outputdir ;\
LockFile = /tmp/kazimir.lock ; EventUpdateInt = 5

Log: Name = PsEdf ; Type = CMND_OUT ; Path = ps -edf | grep toto | grep -v grep ; UpdateInt = 10s ; TimeFormat = NONE


Pattern: Name = PsToto ; Log = PsEdf ; RegExp =toto;

Event : Name = TotoNotHere ; Fenetre = 15s;
Begin
!PsToto
End


Order: Nom = ordre1 ; Event = TotoNotHere ; Action = StartToto;
Action: Nom = StartToto ; Path = /usr/local/bin/toto ;
```

For testing this create a binary 'toto' (during my test is was just a test who ran a 'sleep 5000'). Then run kazimir and look at the log. Then try to kill toto

[8] Yes, I know, I am very found of the word 'toto'....

and see how kazimir restart it. On this example, I have used very short Update value and window time value. They are clearly too short, but they fit well for a demo. So if you want to use a similar event and pattern, use a longer 'UpdateInt' for the PsEdf Log and a longer window size for the event.

A simple usage of variable

I want to look either for Titi and Toto, but want just an action. So, I'll make use of internal variable $(LAST_PATTERN) to define only one action:

```
Kazimir: LogFile = /dev/tty ; EventDir = /tmp/eventdir ; OutputDir = /tmp/outputdir ;\
LockFile = /tmp/kazimir.lock ; EventUpdateInt = 5

Log: Name = syslog ; Type = ASCII ; Path = /var/adm/messages ; TimeFormat=%b %e %H:%M:%S;


Pattern:  Name =Toto; RegExp =toto was here; Log = syslog ;
Pattern:  Name =Titi; RegExp =titi was here; Log = syslog ;

Event : Name =TotoTitiEvent; Window = 30s;
Begin
        Toto || Titi
End

Order: Name = ordre1 ; Event =TotoTitiEvent; Action = Action1 ;
Action: Name = Action1 ; Path = echo "$(LAST_PATTERN) was found" ;
```

A little more complicated: Using a variable counter

This configuration file is close to the previous one. This time I made a single 'echo' to a file to create the log message, this makes it easy for this example to have several time the same message in the log. What we want here is to have kazimir realizes an event every 3 time a pattern is found. So the event will be realized when 3 messages will be found, when 3 others, and so on. This is just an example to show variable in action . Before starting, make sure file /tmp/totolog is empty or does not exists (kazimir will re-open it if necessary).

```
Kazimir: LogFile = /dev/tty ; EventDir = /tmp/eventdir ; OutputDir = /tmp/outputdir ;\
LockFile = /tmp/kazimir.lock ; EventUpdateInt = 5

Log: Name = totolog ; Type = ASCII ; Path = /tmp/totolog ; TimeFormat=NONE

Variable: Name = cpt ; Default = 0 ;

Pattern:  Name =TotoWasHere; RegExp =toto was here; Log = totolog ; Let = { $(cpt) +<- 1 ;} ;

Event : Name = TotoEvent ; Window = 10m; Let = { $(cpt) <- 0 ;} ;
Begin
  TotoWasHere && ( $(cpt) == 3 )
End

Order: Name = ordre1 ; Event = TotoEvent ; Action = Action1;
Action: Name = Action1 ; Path = echo "I have located Toto 3 times" ;
```

Then perform *echo 'toto was here'* once, wait about 5 second then do it again, and so on after a few seconds and see kazimir working. This time the counter works by being used as a boolean element in the event. This is only an example to show how variable can be used

An example with a trigger

This is just a simple example that extends one of the former examples that I gave in this document. It has no other interest than providing a sample of the use of a trigger. There was an example to start a dead process with use of "ps -edf" as a CMND_OUT log. I just added a trigger to say hello to the newly started process. The trigger is activated only the the event that is associated to him triggers from 'event activated' to 'event not activated'.

```
Kazimir: LogFile = /dev/tty ; EventDir = /tmp/eventdir ; OutputDir = /tmp/outputdir ;\
LockFile = /tmp/kazimir.lock ; EventUpdateInt = 5

Log: Name = PsEdf ; Type = CMND_OUT ; Path = ps -edf | grep toto | grep -v grep ; UpdateInt = 10s ; TimeFormat = NONE


Pattern: Name = PsToto ; Log = PsEdf ; RegExp =toto;

Event : Name = TotoNotHere ; Fenetre = 15s;
Begin
!PsToto
End

Trigger: Name = TototIsBack ; Event = TotoNotHere ;

Order: Name = ordre1 ; Event = TotoNotHere ; Action = StartToto;
Action: Name = StartToto ; Path = /usr/local/bin/toto ;

Order: Name = TotoReturns ; Event = TototIsBack ; Action = SayHelloToto ;
Action: Name = SayHelloToto ; Path = echo ‘‘Hello Toto’’
```

An example with a repetition

In this example, repetition R1 is associated with TotoNotHere. If this event is realized 3 times consecutively then the repetition is realized. In this case, the event window time is 15s, and event are checked very 5s. The repetition value is 3, this means that if TotoNotHere is activated during at least 3 x 5s = 15s, the repetition will be activated. This example has a different effect from the one with a variable used as an event counter. In the former example, the event was realized at the third occurrence of a pattern. No matter if the time between the occurrences was 5s of 3 days. The occurrence could be not consecutive and separated by periods when the event was off. This makes this example quite different from the 'Repetition' which imply that all the occurrences are grouped together with no change of the event state.

```
Kazimir: LogFile = /dev/tty ; EventDir = /tmp/eventdir ; OutputDir = /tmp/outputdir ;\
LockFile = /tmp/kazimir.lock ; EventUpdateInt = 5

Pattern: Name = PsToto ; Log = PsEdf ; RegExp =toto;

Event : Name = TotoNotHere ; Fenetre = 15s;
```

```
Begin
!PsToto
End

Repetition: Name = R1 ; Event = TotoNotHere ; Number = 3 ;

Order: Name = ordre1 ; Event = TotoNotHere ; Action = StartToto;
Action: Name = StartToto ; Path = /usr/local/bin/toto ;

Order: Name = Toto3TimeConsecutive ; Event = R1 ; Action = SayHelloToto ;
Action: Name = SayHelloToto ; Path = echo ‘‘Hello Toto’’
```

A few advices about configuration file

This part contains some tips that could be useful when writing kazimir configuration file. If you think things should be added there, just tell me. This part is supposed to be the most dynamic part of the document.

- try to avoid 'NONE' time format with ASCII file, because kazimir reads the whole file just after opening it. So you could find patterns that occurred a long time ago, and this could generate detections of problem that did not occurred in reality.
- when defining the event window time, make sure the Kazimir::EventUpdateInt is shorter than this value. If not, you could miss some problem. I advice you to set up the Kazimir::EventUpdateInt half the minimum value in the event window time that you defined [9]. In a future version of Kazimir, I may automatically adjust the Kazimir::EventUpdateInt that way.

2.6.2 Let's speak french a little

When I wrote Kazimir for the very first time, I did not have the project to make it a free software, and since it was supposed to be used by the guys I am working with (that all are french), all the Kazimir tags and keywords were in french. I made the translation just after, but I wanted this version of kazimir to work also with the former (french-speaking) configuration files already in production. So I kept both the English and french syntax. The table here shows the equivalent that exists for every tag or keyword were the choice is possible. So you can write 'french' kazimir configuration but remember that this feature is mostly there for compatibility reason

[9] for those of you who are a little familiar with 'Signal Acquisition Theory', this is relatively similar to the Shannon's Theorem

Tag or Keyword	French equivalent
Name	Nom
Log	Journal
ASCII	fichier
CMND_OUT	commande
Default	Defaut
Pattern	Motif
Delay	delai
Order	Ordre

Chapter 3

As a conclusion

This is all I had to say about this product. I hope that you found this document both readable and understandable. I am no native English speaker and also the developer of this product, and I have a theory that says you can't find a worse documentation writer for a product than its developer himself. I hope that you will enjoy using kazimir, and that it will fit your need as it fit mines. If you have any comments, suggestions or bug reports, do not hesitate to contact me. By making this tool a free software, I'd like people to give me their opinion about kazimir so that I could make it more and more sophisticated, complete and easier to use.

Contents