Lennart Betz · Thomas Widhalm

Icinga

Monitoring – Grundlagen und Praxis

iX EDITION

dpunkt.verlag

Inhalt

4 Aufbau des eigenen Monitorings

Es ist Zeit, sich nach der Grundkonfiguration und den ersten Gehversuchen auf der Benutzeroberfläche näher mit der Überwachung der Systemlandschaft des Gebrauchtwarencenters zu beschäftigen. Beim Monitoring sollte man sich zuerst immer auf die wirklich wichtigen Systeme und Dienste konzentrieren und nicht auf alles, was möglich ist.

Dennoch erfolgt in der Regel zu Beginn eine Konsolidierung, welche Systeme existieren und wo die wichtigen Dienste laufen. Das erste Ziel sollte sein, alle Geräte oder virtuellen Maschinen als Hosts im Monitoring zu konfigurieren und auf Verfügbarkeit zu überwachen. Ist dies nicht in überschaubarer Zeit möglich, kümmert man sich auch schon früher um die wichtigsten Dienste. In der Regel sind das die Dienste, mit denen ein Unternehmen Geld einnimmt, und solche, die für den reibungslosen Geschäftsbetrieb unverzichtbar sind.

»Ich funktioniere im Rahmen normaler Parameter.«

Lt. Cmdr. Data, 1998

Die IT-Umgebung des Gebrauchtwarencenters

Beim Gebrauchwarencenter Jonas muss das Shop-System für den Onlinehandel rund um die Uhr erreichbar sein, aber auch das Kassensystem vor Ort hat zu den Öffnungszeiten zwischen 9 und 17 Uhr zu laufen.

Bei dem Onlineshop handelt es sich um eine Webseite *shop.gwc-jonas.net*, die auf einem Linux-Server *antlia* als »Virtual Named Host« läuft. Auf dem gleichen Webserver unter derselben IP-Adresse ist auch der Webauftritt *www.gwc-jonas.net* des Gebrauchtwarencenters erreichbar.

Beide Webseiten greifen auf Datenbanken zu, die auf einem weiteren Linux-Server *aquarius* laufen. Bei dem dort installierten DBMS handelt es sich um ein PostgreSQL.

Host	Netzwerk	Plattform	Services
draco	DMZ, LAN	Linux	DNS, DHCP, NTP
kmw	DMZ	Linux	SMTP, ClamAV, Amavis, Squid
gmw	LAN	Windows	Exchange, SMTP, IMAP
sculptor	DMZ	Linux	Icinga
antlia	DMZ	Linux	HTTP, HTTPS
aquarius	LAN	Linux	PostgreSQL
phoenix	LAN	Solaris	ERP, Oracle
fornax	LAN	Linux	Icinga, MySQL
andromeda	LAN	Windows	AD, Terminalserver
carina	LAN	Appliance	Kassensystem
sagittarius	LAN	Linux	Squid
triagulum	LAN	Linux	VMware VCenter
sextans	LAN	Linux	PuppetDB u. Server
canis	LAN	Linux	Elasitc-Stack

Tabelle 4-1: Hosts und Services des Gebrauchtwarencenters

Der Shop kommuniziert zur Bestellabwicklung, Lagerhaltung und Rechnungsstellung ebenfalls mit einem Enterprise Resource Planning (ERP) System und der zugehörigen Oracle-Datenbank auf einem Solaris-Server *phoenix*. Bestellbestätigungen und Rechnungen müssen per Mailversand verschickt werden und benötigen das Relay auf *kmw*.

Das Kassensystem ist eine Appliance, die keinen direkten Zugriff auf ihr laufendes System gestattet. Hier gilt aber die Einschränkung, dass die Kasse außerhalb der Öffnungszeiten ausgeschaltet ist und zu diesen Zeiten nicht überwacht werden kann und soll. Das Gleiche gilt für den Drucker, über den Rechnungen ausgestellt werden, der aber auch die sonstigen Ausdrucke des kleinen Büros bewerkstelligt.

Die im Monitoring nicht zu berücksichtigenden Büroarbeitsplätze mit ihren Windows-Desktop-Systemen sind an ein AD auf *andromeda* angeschlossen, das auch als Server für diverse Terminals dient, über die die Mitarbeiter auf das ERP zugreifen. Als Mail- und Groupware-Server kommt ein Exchange-Server *gmw* zum Einsatz. Während alle Windows-Server 24 Stunden am Tag laufen, werden die Terminals, ebenso wie die Kasse und der Drucker, nur zu den Öffnungszeiten betrieben.

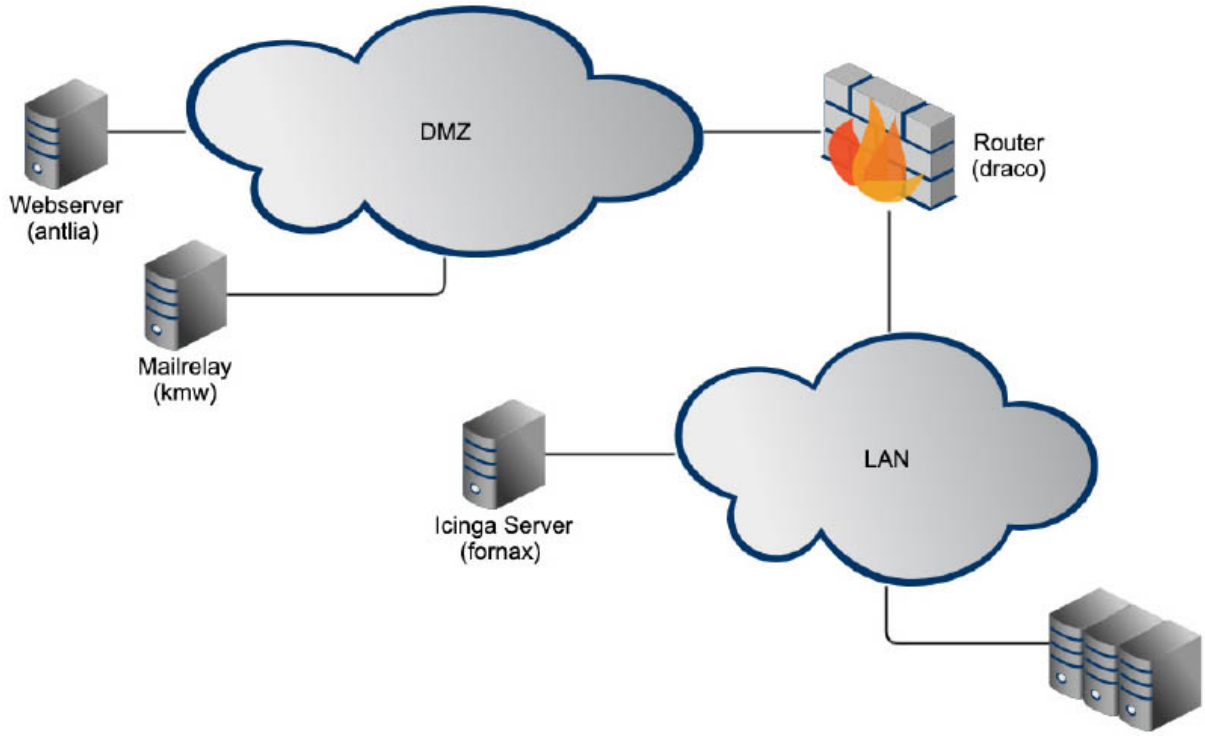

Abbildung 4-1: Logischer Netzwerkplan

Das Netzwerk ist in zwei Segmente unterteilt: eine Demilitarized Zone (DMZ) mit den Hosts *antlia, kmw* sowie dem zu einem späteren Zeitpunkt für Icinga eingeplanten Server *sculptor* und das Local Area Network (LAN) mit allen übrigen Systemen wie z. B. dem Icinga-Server selbst.

Getrennt sind beide Netze durch den Router *draco*, der neben der Internetanbindung auch für beide Netze die Namensauflösung, die dynamische Zuweisung von IP-Adressen via Dynamic Host Configuration Protocol (DHCP) und einen Zeitserver bereitstellt.

Die DMZ hat eine eigene vom LAN *gwc-jonas.local* getrennte Domain *gwc-jonas.net*.

Zu einem späteren Zeitpunkt ist für die DMZ ein eigenes als Satellitensystem arbeitendes Icinga auf einem dedizierten Host *sculptor* vorgesehen, der stellvertretend für den eigentlichen Server Überwachungen ausführen wird.

Vorbereitungen

In diesem Kapitel werden wir einen Blick auf die Beispielkonfiguration werfen, die sich im Verzeichnis *conf.d* unterhalb vom Hauptkonfigurationsverzeichnis */etc/icinga2* befindet, und ebenfalls unterhalb, in *zones.d/main*, unsere eigene Konfiguration zusammenbauen.

> Pfadangaben bei Konfigurationsdateien, die nicht absolut angegeben sind, beziehen sich ab nun immer relativ zu */etc/icinga2*.

Dazu entfernen wir zuerst die Datei, die die Beispielkonfiguration einbindet, und führen einen Reload von Icinga aus, siehe »Beispielkonfiguration« Abschnitt 1.3.2 ab Seite 18.

```
$ cd /etc/icinga2

$ rm zones.d/main/default.conf

$ cp -p conf.d/templates.conf zones.d/main/

$ systemctl reload icinga2
```

Aus den Beispielen unterhalb von *conf.d* bietet sich zunächst die Übernahme der Datei *templates.conf* an, da sie einiges Grundlegendes zur allgemeinen Verwendung enthält.

Bei Konfigurationsdateien ist immer darauf zu achten, dass der Benutzer, unter dem Icinga ausgeführt wird, die Dateien lesen kann. Eine Konfigurationsüberprüfung gibt Auskunft, falls Icinga eine Datei nicht lesen kann:

```
$ icinga2 daemon –validate

…

information/cli: Icinga application loader (version: 2.13.2)

information/cli: Loading configuration file(s).
```

```
warning/ConfigCompiler: Cannot compile
'zones.d/main/templates.conf'


    (0) Compiling configuration file
    'zones.d/main/templates.conf'


information/ConfigItem: Committing config item(s).


…
```

Darüber hinaus enthält die Ausgabe weitere nützliche Informationen, neben der Icinga-Version auch die Anzahl der jeweiligen Objekte der aktuellen Konfiguration nach Typen aufgeteilt.

Auch **Fehler werden hier**, wenn der Start oder Reload fehlschlägt, **aussagekräftig ausgegeben**.

4.1 Kleine Sprachreferenz Icinga-DSL

Eine Domain Specific Language ist eine auf ein bestimmtes Anwendungsfeld (die sog. Domäne) zugeschnittene formale Sprache. Im Falle von Icinga wird beschrieben, dass etwas überwacht werden soll, es wird aber nicht festgelegt, wie das zu geschehen hat. Die eigentliche Überwachung übernehmen dann die Plugins.

Leider basiert die Icinga-DSL auf keiner anderen Sprache, wie z. B. die auf Ruby basierenden Puppet-DSL und Rake, weshalb Sie zunächst eine neue Sprache erlernen müssen. Als DSL ist sie jedoch kompakt und übersichtlich und damit leicht zu erlernen.

»Ich spreche nur meine eigene Sprache und damit hab ich schon Probleme.«

Korben Dallas, 1997

4.1.1 Konstanten

In der Datei */etc/icinga2/constants.conf* sind einige Konstanten definiert, die in anderen Konfigurationsdateien verwendet werden können. Hier ein beispielhafter Ausschnitt:

```
const PluginDir = "/usr/lib64/nagios/plugins"

//const NodeName = "localhost"
```

Codebeispiel 4.1: */etc/icinga2/constants.conf*

Standardmäßig ist *PluginDir* vordefiniert und legt den Pfad auf die sogenannten »Standard-Plugins« fest. Je nachdem, welche Distribution Sie verwenden, ist hier ein anderer Wert voreingestellt (das obige Beispiel stammt von einem RedHat-System). Unter Debian oder einem Derivat befinden sich die Monitoring-Plugins in `/usr/lib/nagios/plugin`.

Der *NodeName* definiert den Namen des Servers. Im obigen Beispiel ist die entsprechende Zeile durch die beiden Slashes auskommentiert und daher nicht gesetzt. In diesem Fall wird bei einem korrekt konfigurierten Server automatisch der Fully Qualified Domain Name (FQDN) verwendet.

4.1.2 Objekt- und Template-Definition

Die gesamte Konfiguration ist objektbasiert. Objekte werden mit dem Schlüsselwort *object* definiert, gefolgt von einer frei wählbaren Bezeichnung für dieses Objekt. Der Name muss konfigurationsweit eindeutig für Hosts sein. Typ und Name bilden zusammen ein eindeutiges identifizierendes Merkmal eines Objekts.

Hier ist ein Ausschnitt eines Objekts vom Typ *Host* aus der Beispielkonfiguration:

```
object Host NodeName {

  import "generic-host"

  address = "127.0.0.1"
```

```
  address6 = "::1"

  vars.os = "Linux"

  vars.disks["disks"] = {}

  vars.disks["disk /"] = {

    disk_partition = "/"

  }

}
```

Codebeispiel 4.2: Beispiel eines Hostobjekts *conf.d/hosts.conf*

Als Name ist sogleich auch eine Konstante in Verwendung. In der Beispielkonfiguration entspricht dieser Host damit dem Icinga-Server selbst. Bei *address* und *address6* handelt es sich um Attribute. Attribute sind objektspezifisch, diese beiden gelten zum Beispiel nur für Objekte vom Typ *Host*.

Im Gegensatz dazu können Custom Variables frei vergeben und benannt werden. Während *address* ein fester Bestandteil eines Hostobjekts ist, handelt es sich bei *os* um eine solche Custom Variable. Sie haben keine spezielle Bedeutung für Icinga 2 selbst und werden im Attribut *vars* als Key-Value-Paar gespeichert. Custom Variables können für Informationen zu Objekten, insbesondere aber für Regeln verwendet werden, die Objekte miteinander verbinden, wie im weiteren Verlauf immer wieder gezeigt werden wird.

Jedes Objekt hat einen festen Satz an Attributen und kann beliebig viele Custom Variables erhalten. Viele Nicht-Custom-Variables sind optional und können weggelassen werden. Eine Übersicht, welche Attribute ein Objekt hat, findet sich in der Onlinedokumentation zu Icinga 2. Eine genauere Beschreibung von Custom Variables folgt noch in diesem Kapitel.

```
template Host "generic-host" {
```

```
  max_check_attempts = 3

  check_interval = 1m

  retry_interval = 30s

  check_command = "hostalive"

}
```

Codebeispiel 4.3: Generisches Template für Hosts *conf.d/templates.conf*

Zusätzlich können von jedem Objekttyp auch Templates angelegt werden. Diese werden als Vorlagen mit *import* zu einem Objekt gleichen Typs hinzugeladen. Die dort gesetzten Attribute und Custom Variables werden mit ihren Werten dann dem eigentlichen Objekt hinzugefügt. Dort können die ererbten Werte optional auch wieder überschrieben werden.

4.1.3 Datentypen

Folgende Ausdrücke können durch eine Zuweisung mit »=« auf der rechten Seite verwendet werden. Bei Konstanten und Custom Variables ist der Wert bzw. dessen Typ im Gegensatz zu Attributen nicht festgelegt. Attribute haben einen festen Einsatzzweck und sind zumeist an einen Datentyp bei ihrer Zuweisung gebunden.

Floating Point als Zahl in angelsächsischer Notation mit Punkt.

```
a = 7.3
```

Floating Point als »sprechende« Darstellung bei Zeitangaben mit *ms* (Millisekunden), *s* (Sekunden), *m* (Minuten), *h* (Stunden) und *d* (Tage). Ist keine Einheit angegeben, wird Sekunden angenommen.

```
a = 1.5m
```

Strings; die Zeichen in Tabelle 4-2 benötigen eine Escape-Sequenz.

```
a = "Hello World!"
```

Zeichen	Escape-Sequenz
”	\”
\	\
$	$$
<TAB>	\t
<CARRIAGE-RETURN>	\r
<LINE-FEED>	\n
<BEL>	\b
<FORM-FEED>	\f

Tabelle 4-2: Zeichen und ihre Escape-Sequenz

Multiline Strings, die sich über mehrere Zeilen erstrecken, sind wie folgt zu klammern:

```
a = {{{Hello

all

over the

world!}}}
```

Boolean mit den Werten *true* und *false*.

Null Value; mit dem Schlüsselwort *null* wird ein leerer Wert angegeben.

Arrays sind eine geordnete, kommaseparierte Liste unterschiedlicher Elemente. Die Liste darf sich auf mehrere Zeilen verteilen.

```
a = [ "the question", 42 ]
```

Dictionary ist eine ungeordnete Liste von Key-Value-Paaren. Die Schlüssel müssen eindeutig sein. Der Vergleich der Schlüssel ist case-sensitiv. Die Paare sind mit Komma voneinander zu trennen oder jeweils in eine separate Zeile zu schreiben.

```
a = {

  address = "172.16.1.11"

  port = 443

}
```

Die einzelnen Elemente eines Dictionarys lassen sich auf unterschiedliche Weisen ansprechen oder zuweisen. So kann ein Schlüssel durch die Trennung mit einem Punkt zum Dictionary angesprochen werden oder als Index mit eckigen Klammern.

```
dictionary.key = value

dictionary["key"] = value
```

Letzteres ermöglicht auch die Verwendung von Leerzeichen in der Schlüsselbezeichnung. Der Zuweisungsoperator »+=« fügt das Paar additiv dem Dictionary hinzu oder überschreibt dieses bei Vorhandensein mit dem neuen Wert.

```
dictionary += {

  key = value
```

```
}
```

4.1.4 Funktionen

Die Icinga-DSL kennt ebenfalls Funktionen. Eine große Anzahl kommt bereits im Standard des Sprachumfangs mit und kann in einer Gesamtübersicht in der Onlinedokumentation[1] eingesehen werden.

Es gibt Funktionen, die auf Strings, Arrays und Dictionarys operieren, mathematische, boolesche und welche, die komplette Icinga-Objekte oder Referenzen auf diese zurückliefern.

Einige werden in diesem Buch sukzessive eingeführt und an den entsprechenden Stellen gesondert vorgestellt.

4.2 Host und Hostgruppen

Das erste Teilziel, alle Geräte als Hosts ins Monitoring aufzunehmen, legt nahe, sich zuerst mit Objekten vom Typ Host zu beschäftigen. In der Datei `conf.d/hosts.conf` befindet sich das Hostobjekt *NodeName*, ein Host mit dem Namen des Werts der gleichnamigen Konstante aus `constants.conf`. Die Attribute *address* und *address6* bilden mit den ererbten Attributen aus *generic-host* zusammen die Eigenschaften des Objekts. Zusätzlich sind die Eigenschaften über das Dictionary *vars* noch um die Custom Variable *os* erweitert.

```
object Host NodeName {

  import "generic-host"

  address = "127.0.0.1"

  address6 = "::1"
```

```
  vars.os = "Linux"

}
```

Codebeispiel 4.4: Hostdefinition in *conf.d/hosts.conf*

In *conf.d/templates.conf* findet sich die Definition des eben benutzten Templates *generic-host*. Hostobjekte, die dieses Template einbinden, bekommen die hier gesetzten Attribute automatisch als Standardwerte zugewiesen.

Mit *check_interval* wird der Zeitabstand angegeben, in dem der Host auf Erreichbarkeit geprüft werden soll. Das Attribut *retry_interval* setzt dieses Intervall auf den angegebenen Wert herab, für den Fall, dass Icinga einen Statuswechsel von UP auf DOWN feststellt.

```
template Host "generic-host" {

  max_check_attempts = 3

  check_interval = 1m

  retry_interval = 30s

  check_command = "hostalive"

}
```

Codebeispiel 4.5: Template *generic-host* in *zones.d/main/templates.conf*

Das nun reduzierte Intervall wird für die in *max_check_attempts* angegebene Anzahl von Checks beibehalten, bevor es wieder auf das »normale« Intervall zurückfällt. Die Checks werden demnach im Fall eines erkannten Fehlers für eine gewisse Zeit strenger überwacht, um sicherzustellen, dass es sich nicht nur um einen kurzfristig auftretenden Fehler handelt, für den eine Alarmierung nicht nötig ist.

Diese drei Attribute spielen auch für die Benachrichtigung eine wichtige Rolle, wie in Abschnitt 9.1 ab Seite 182 erläutert wird. Das Check Command *hostalive*

ist vordefiniert in Icinga enthalten und beruht auf dem Plugin `check_ping`.

Mit diesen Mitteln lässt sich das Zwischenziel, alle Geräte auf Verfügbarkeit zu prüfen, bereits erreichen. Da das Gebrauchtwarencenter sowohl einige Linux- wie auch mehrere Windows-Systeme sein eigen nennt, bietet sich die Definition zweier Templates an. Das Template *linux-host* kommt in die Datei *zones.d/main/linux.conf* und *windows-host* für Windows-Systeme in *zones.d/main/windows.conf*.

```
/*

 * Host Templates

 */

template Host "linux-host" {

  import "generic-host"

  vars.os = "Linux"

}
```

Codebeispiel 4.6: *zones.d/main/linux.conf*

Im Verlauf werden wir die Templates, aber vor allem die beiden Dateien erweitern, und das nach der logischen Unterteilung Linux – Windows.

```
/*

 * Host Templates

 */

…
```

```
template Host "windows-host" {

  import "generic-host"

  vars.os = "Windows"

}
```

Codebeispiel 4.7: *zones.d/main/windows.conf*

Um nun weitere Hosts wie z. B. für den Webserver *antlia* in die Überwachung aufzunehmen, wird einfach ein Hostobjekt in einer neuen Datei *zones.d/main/hosts/antlia.conf* hinzugefügt. Damit auch ein Check ausgeführt wird, muss dem neuen Objekt eine IP-Adresse im Attribut *address* oder in *address6* mitgegeben werden.

```
object Host "antlia.gwc-jonas.net" {

  import "linux-host"

  display_name = "antlia"

  address = "172.16.2.12"

}
```

Codebeispiel 4.8: Neuen Host in *zones.d/main/hosts/antlia.conf* anlegen

Mit *display_name* wird die Bezeichnung der Anzeige in Icinga Web 2 angepasst. Wird das Attribut *display_name* nicht verwendet, ist der Anzeigename identisch mit dem Objektnamen.

Analog wird bei allen anderen Linux- und Windows-Hosts verfahren, natürlich bis auf den Unterschied, welches Template zu verwenden ist. Die Solaris-Maschine *phoenix* ist als Unix-System einem Linux sehr ähnlich und so kann auch hier das Template *linux-host* benutzt werden, die Custom Variable *os* wird allerdings überschrieben:

```
object Host "phoenix.gwc-jonas.local" {

  import "linux-host"

  display_name = "phoenix"

  address = "172.16.1.15"

  vars.os = "Solaris"

}
```

Codebeispiel 4.9: *zones.d/main/hosts/phoenix.conf*

Der Hintergedanke hierbei ist, auch im weiteren Verlauf des Konfigurationsausbaus von Informationen aus dem Linux-Template zu profitieren. Alle anderen Geräte, wie die Switche des Gebrauchtwarencenters, arbeiten zunächst mit dem generischen Template *generic-host*.

Hosts lassen sich zu Hostgruppen zusammenfassen. Zwei solche Gruppendefinitionen stehen in *conf.d/groups.conf*. In beiden wird ein neues und mächtiges Mittel von Icinga 2 benutzt, der *assign*-Filter. Ist der boolesche Ausdruck für die Objekte in *assign where* wahr, werden diese in die Gruppe aufgenommen. Im folgenden Beispiel sind das alle Hostobjekte, die die Custom Variable *os* entsprechend gesetzt haben.

```
object HostGroup "linux-servers" {

  display_name = "Linux Servers"

  assign where host.vars.os == "Linux"

}

object HostGroup "windows-servers" {

  display_name = "Windows Servers"
```

```
  assign where host.vars.os == "Windows"

}
```

Codebeispiel 4.10: Definition von Hostgruppen in *conf.d/groups.conf*

Hostgruppen sind im Folgenden keine Notwendigkeit, können aber gerne in die jeweilige Datei *linux.conf* und *windows.conf* übernommen werden. Am besten versieht man die Zeilen davor mit einem Kommentar wie bei den Templates.

Bei Anpassungen an der Konfiguration der zu überwachenden Objekte muss Icinga 2 diese mit einem einfachen Reload neu einlesen.

```
$ systemctl reload icinga2
```

Danach erscheinen die neuen Hosts in Icinga Web 2 zunächst mit dem Zustand PENDING bis zum ersten Check, ab dann müssen sie UP oder DOWN anzeigen.

4.3 Service und Servicegruppen

Ein Service muss stets genau einem Host zugeordnet sein. Diese Zuordnung kann über das Attribut *host_name* erfolgen, Icinga 2 bietet aber eine wesentlich elegantere Methode. Bei einem Blick in *conf.d/services.conf* wird für die Services *ping4* bzw. *ping6* eine *apply*-Regel verwendet. Diese erzeugt mithilfe der schon bekannten *assign*-Filter je einen Service mit angegebenen Namen zu allen Hostobjekten, die den booleschen Ausdruck mit »wahr« erfüllen.

Die beiden genannten Services können eins zu eins in die neue Datei für Netzwerkbelange *zones.d/main/network.conf* übernommen werden.

```
/*

 * Services

 */
```

```
apply Service "ping4" {

  import "generic-service"

  check_command = "ping4"

  assign where host.address

}

apply Service "ping6" {

  import "generic-service"

  check_command = "ping6"

  assign where host.address6

}
```

Codebeispiel 4.11: Service mittels *apply* in *zones.d/main/network.conf*

Allen Hostobjekten, die das Attribut *address* definiert haben, wird auch der Service *ping4* zugewiesen. Analoges gilt für *address6* und *ping6*. Das importierte Template befindet sich bereits in der Datei *templates.conf* im Verzeichnis *zones.d/main*. Die Attribute haben dieselben Bedeutungen wie beim Hostobjekt. Die benötigten Check Commands sind standardmäßig in Icinga vorhanden und beruhen ebenfalls auf dem Plugin check_ping, allerdings mit deutlich niedrigeren Schwellenwerten.

```
template Service "generic-service" {

  max_check_attempts = 5

  check_interval = 1m
```

```
  retry_interval = 30s

}
```

Codebeispiel 4.12: Template *generic-service* in *zones.d/main/templates.conf*

Zurück zur Datei conf.d/services.conf. Dort befindet sich auch die Definition, die einen Service *ssh* an allen als Linux klassifizierten Hosts erzeugt, die zusätzlich die Attribute *address* oder *address6* gesetzt haben müssen:

```
apply Service "ssh" {

  import "generic-service"

  check_command = "ssh"

  assign where (host.address || host.address6) \

      && host.vars.os in ["Linux, "Solaris"]

}
```

Codebeispiel 4.13: Apply-Service *ssh* in *zones.d/main/unix.conf*

Auch dieser Service kann mit einer kleinen Ergänzung für Solaris-Hosts übernommen werden. Der Operator *in* prüft, ob sich der Inhalt der Custom Variable *os* mit einem Element der nachfolgenden Liste deckt. Da es sich nun nicht mehr um einen rein Linux-spezifischen Service handelt, gehört er folgerichtig in eine Datei *zones.d/main/unix.conf* gespeichert.

Auch Services lassen sich gruppieren. Die folgende Definition ist in *conf.d/groups.conf* abgelegt und verwendet die Funktion *match* als *assign*-Filter. Die Servicegruppe *ping* enthält alle Services, deren Namen mit dem String »ping« beginnen.

```
object ServiceGroup "ping" {
```

```
  display_name = "Ping Checks"

  assign where match("ping*", service.name)

}
```

Codebeispiel 4.14: Servicegruppe mittels einer Funktion in *conf.d/groups.conf*

Wer auch diese Gruppe übernehmen möchte, dem sei als Ablageort die Datei *zones.d/main/network.conf* empfohlen.

Service mittels Dictionary automatisch erzeugen

Die Beispielkonfiguration hält noch eine weitere wichtige Eigenschaft der DSL bereit, nämlich einen definierten Service auf ein und demselben Host mehrfach zu verwenden. Die Services unterscheiden sich natürlich im Namen, aber auch die Parametrisierung ist unterschiedlich wählbar.

So ist das Überwachen von unterschiedlichen Webseiten mit individuellem Uniform Resource Identifier (URI) oder Schwellenwerten möglich. Im Hostobjekt *NodeName* in *conf.d/hosts.conf* findet sich ein Beispiel hierzu.

```
vars.http_vhosts["http"] = {

  http_uri = "/"

}

/* Uncomment if you've sucessfully installed Icinga Web 2. */

vars.http_vhosts["Icinga Web 2"] = {

  http_uri = "/icingaweb2"

}
```

Codebeispiel 4.15: Webseiten auf dem Host *NodeName* in `conf.d/hosts.conf`

Hier ist eine Custom Variable *http_vhosts* als Dictionary definiert und dessen Elementen *http* und *Icinga Web 2* ist als Wert jeweils wieder ein Dictionary zugewiesen. Diese enthalten unterschiedliche Parameter, die einen unterschiedlichen Plugin-Aufruf zur Folge haben.

```
apply Service for (http_vhost => config in
host.vars.http_vhosts) {

  import "generic-service"

  check_command = "http"

  vars += config

}
```

Codebeispiel 4.16: Service *http* in `zones.d/main/webserver.conf`

Das Serviceobjekt für diese Checks offenbart sein Geheimnis in der Datei `conf.d/services.conf`. Neben der *apply*-Regel findet sich eine *for*-Schleife in der Definitionszeile.

Jedes Element aus *vars.http_vhosts* zu jedem Host, der diese Custom Variable enthält, wird von *for* durchlaufen und es wird jeweils ein Service erzeugt. Der Name entspricht hierbei den jeweiligen Schlüsselnamen von *http_vhosts*. Der Teil *http_vhost => config in host.vars.http_vhosts* bedeutet, dass der Schlüsselname, im zweiten Beispiel *Icinga Web 2*, in jedem Schleifendurchlauf über die Variable *vhost_http* referenziert werden kann. Ebenso kann der Wert, im zweiten Beispiel das Dictionary, das nur *http_uri = /icingaweb2* enthält, als *config* angesprochen werden.

Serviceobjekte haben ihr eigenes von Hostobjekten getrenntes Dictionary *vars* für Custom Variables. Diesem wird mit der letzten Zeile das Dictionary *config* mit den enthaltenen Parametern für den Plugin-Aufruf des Services hinzugefügt. Das gesamte Dictionary wird also mit dem *vars*-Dictionary des Services zusammengelegt und jedes Element gilt für Icinga 2 so, als wäre es im Service

mit *vars* direkt definiert worden. Wäre das zweite Beispiel ohne *apply for* angelegt worden, müsste er deshalb *vars.http_uri = /icingaweb2* enthalten.

Es wäre möglich, wenn auch unüblich, einzelne Elemente des Dictionarys zu übernehmen. Statt *vars += config* wäre dann *vars.http_uri = config.http_uri* in den Service aufzunehmen. www komplette Übernahme des Dictionarys bietet aber deutlich mehr Flexibilität, da eine zusätzliche Konfiguration nur beim Host hinzugefügt werden muss.

Mit der folgenden einfachen Erweiterung werden z. B. die Schwellenwerte für Antwortzeiten (in Sekunden) der Seite gesetzt.

```
vars.http_vhosts["Icinga Web 2"] = {

  http_uri = "/icingaweb2"

  http_warn_time = 1

  http_crit_time = 2

}
```

Genau so funktioniert auch die Überwachung von Dateisystemen in der Beispielkonfiguration, dazu später mehr in Abschnitt 6.4 ab Seite 116.

Um nun die Webseiten des Gebrauchtwarencenters zu überwachen, ist die Servicedefinition mittels *apply* nach *zones.d/main/network.conf* zu übernehmen und dem Objekt zum Host *antlia* zwei solcher Custom Variables hinzuzufügen, eine für den Webauftritt der Firma und eine weitere, die den Onlineshop überwacht.

```
vars.http_vhosts["Website"] = {

  http_vhost = "www.gwc-jonas.net"

}

vars.http_vhosts["Online Shop"] = {
```

```
  http_vhost = "shop.gwc-jonas.net"

  http_ssl = true

}
```

Codebeispiel 4.17: Ergänzungen in *zones.d/main/hosts/antlia.conf*

Beide Webseiten werden über denselben Host mit dem Verfahren »Virtual Named Host« ausgeliefert, deshalb muss der Name der Seite jeweils mit *http_vhost* angegeben werden. Da der Shop nur verschlüsselt erreichbar ist, muss zusätzlich mit *http_ssl* für diesen Service das Protokoll HTTPS ausgewählt werden.

Eine weitaus detailreichere Einführung in die Überwachung von Webseiten und deren Server ist in Abschnitt 14.5 ab Seite 299 zu finden.

4.4 Check Commands und die Template Library

Check Commands stellen das Bindeglied zum Plugin dar. Sie regeln, wie ein Plugin zur Ausführung gelangt und mit welchen Optionen es aufgerufen wird. Die bisher eingesetzten Check Commands *hostalive, check_ping* und *check_http* wurden als Bestandteil von Icinga 2 bezeichnet, was nicht ganz richtig ist. Sie sind jedoch Teil der ITL, die mit Icinga 2 ausgeliefert wird.

Bei der ITL handelt es sich eher um eine umfangreiche Sammlung von Check-Command-Definitionen anstatt um Templates. Sie befindet sich in */usr/share/icinga2/include* und besteht aus mehreren unterschiedlichen Dateien, die sich ggf. gegenseitig einbinden. So erfolgt in *plugins.conf* das Laden von *command-plugins.conf* unter Zuhilfenahme von *include*. In dieser Datei befinden sich Check-Command-Definitionen zu den Plugins aus dem Monitoring-Plugins-Projekt[2].

```
include "command-plugins.conf"
```

Codebeispiel 4.18: */usr/share/icinga2/include/plugins*

Auf diese Weise werden Check Commands für häufig verwendete Plugins von Icinga 2 gleich mitgeliefert. Im Gegensatz zu anderen Teilen der Konfiguration gibt es nicht viel Wahlmöglichkeiten in diesen Definitionen, außerdem sind sie oft lang und eintönig. Durch die fertig ausformulierten Check Commands in der ITL nimmt das Icinga-Team den Anwendern viel Arbeit im Alltag ab.

Es lohnt sich, immer wieder einen Blick in die ITL zu werfen, da sie mit neuen Versionen von Icinga 2 regelmäßig erweitert wird. Eigene Check Commands sollte man übrigens keinesfalls der ITL hinzufügen, da diese von den Paketen nicht als Konfigurationsdatei angesehen und bei Updates einfach überschrieben werden. Besser aufgehoben sind sie stattdessen in einer eigenen selbstverwalteten Datei.

Als Beispiel eines Check Commands dient hier das in der ITL abgelegte Kommando *load*.

```
object CheckCommand "load" {

  command = [ PluginDir + "/check_load" ]

  arguments = {

    "-w" = {

      value = "$load_wload1$,$load_wload5$,$load_wload15$"

      description = "Exit with WARNING status if load …"

    }

    "-c" = {

      value = "$load_cload1$,$load_cload5$,$load_cload15$"

      description = "Exit with CRITICAL status if load …"
```

```
    }

    "-r" = {

      set_if = "$load_percpu$"

      description = "Divide the load averages by the …"

    }

  }

  vars.load_wload1 = 5.0

  vars.load_wload5 = 4.0

  vars.load_wload15 = 3.0

  vars.load_cload1 = 10.0

  vars.load_cload5 = 6.0

  vars.load_cload15 = 4.0

  vars.load_percpu = false

}
```

Codebeispiel 4.19: Check Command *load* aus der ITL-Sammlung

In der Option *command* wird im ersten Element des erwarteten Arrays der Pfad zum zugehörigen Plugin angegeben. Hier setzt er sich aus zwei Strings zusammen, wobei es sich bei *PluginDir* um eine Konstante aus *constants.conf* handelt, die den konkreten Pfad enthält. Weitere Array-

Elemente werden beim Aufruf des Plugins jeweils durch Leerzeichen getrennt als zusätzliche Optionen mitgegeben, d. h., wird ein Element vor den Pfad zum Plugin gesetzt, steht es auch im Aufruf davor!

Das Herzstück des Objekttyps *CheckCommand* ist das Dictionary *arguments*. Es beschreibt die Optionen des Plugins und deren Verhalten. So ist die Option `-w` im Aufruf nur gesetzt, wenn die Auswertung der Makros *$load_wload1$, $load_wload5$* und *$load_wload15$* für *value* einen Wert ergeben. Ist das der Fall, wird dessen Wert hinter der Option eingefügt. Handelt es sich bei der Option, wie bei `-r`, nur um einen Schalter ohne zugehörigen Wert, verwendet man anstatt *value* den Schlüssel *set_if*. Ein beschreibender Text zur jeweiligen Option sollte mittels *description* optional hinzugefügt werden. Er hat jedoch in den bis zum Erscheinen des Buchs veröffentlichten Versionen keinerlei Funktion.

Eine weitere Option im *arguments*-Dictionary ist *required*. Damit wird deutlich gemacht, dass die zugehörige Plugin-Option zwingend benötigt wird. Mit *skip_key* wird die im Schlüssel angegebene Option beim Aufruf des Plugins nicht mit eingesetzt. Dies wird bei Plugins eingesetzt, die zwar Argumente benötigen, aber dazu keine Optionen benutzen.

Name	**Typ**	**Beschreibung**
value	String/Function	**optional** Argumentenwert als Makro oder Funktion
key	String	**optional** überschreibt die Option
description	String	**optional** eine Beschreibung des Arguments
required	Boolean	**false** (default) erforderliches Argument
skip_key	Boolean	**false** (default) unterdrückt die Option
set_if	String/Function	**optional** Option ohne Wert, wenn Makro definiert
order	Number	Setzen der Reihenfolge in der Argumentenliste
repeat_key	Boolean	**true** (default) Option wird mehrfach verwendet, wenn *value* ein Array ist

Tabelle 4-3: Verwendbare Keys im Argumenten-Dictionary

Genau bei solchen Plugins ist die Reihenfolge der Argumente meistens ebenfalls entscheidend. Ein Dictionary ist aber ein unsortierter Key-Value-Store und

deshalb existiert mit *order* die Möglichkeit, die Reihenfolge der Argumentenliste zu bestimmen. Argumente mit demselben Wert werden untereinander unsortiert angefügt, der Default ist »0«. Danach folgen in aufsteigender Reihenfolge die Argumente mit positiven Werten. Negative Werte sind ebenfalls möglich und reihen Argumente vor denen mit *order* gleich »0« ein. Das bewirkt hier: Je kleiner der Wert, desto weiter »vorne« steht das Argument in der Liste.

```
#!/bin/bash

if [ "$2" == "–fqdn" ]; then

  HOSTNAME=$(hostname –fqdn) || exit 3

else

  HOSTNAME=$(hostname –short) || exit 3

fi

if [ $HOSTNAME == $1 ]; then

  echo "OK: hostname matches $1" && exit 0

else

  echo "WARNING: hostname does not match $1" && exit 1

fi
```

Codebeispiel 4.20: Shell-basiertes Plugin `check_hostname`

Müssen bei einem Plugin mehrere Argumente zu einer Option angegeben werden, wird dies mit der Übergabe eines Arrays an *value* gemacht. Nun kann es möglich sein, dass das Plugin die zugehörige Option entweder nur einmal fordert oder vor jedem Element. Aus dem Check Command heraus steuert das

Icinga 2 mit *repeat_key*. Standardmäßig ist *true* gesetzt und die Option wird mehrfach verwendet.

Damit ist es an der Zeit, ein eigenes Check Command zu schreiben. Aber zu welchem Plugin? Am besten gleich zu einem eigenen. Ein einfaches Shell-basiertes Plugin zur Ermittlung des eigenen Hostnamens könnte aussehen wie in Codebeispiel 4.20.

Das Plugin `check_hostname` ohne ausreichende Fehlerbehandlung fordert als ersten Parameter einen String, der im Folgenden mit dem ermittelten Hostnamen verglichen wird. Stimmen beide überein, wird ein OK, andernfalls ein WARNING zurückgegeben. Mit der Option `–fqdn` bezieht sich der Vergleich auf den FQDN.

```
object CheckCommand "hostname" {

  command = [ MyPluginDir + "/check_hostname" ]

  arguments = {

    "–match" = {

      value = "$hostname_match$"

      skip_key = true

      order = 1

    }

    "–fqdn" = {

      set_if = "$hostname_fqdn$"

      order = 2

    }
```

```
  }

  vars.hostname_match = "$host.name$"

}
```

Codebeispiel 4.21: Check Command *hostname* in *zones.d/main/linux.conf*

Soll das Plugin verständlicherweise nicht bei den Monitoring-Plugins nach *PluginDir* abgelegt werden, kann eine neue Konstante *MyPluginDir* in `constants.conf` auf den eigenen Pfad gesetzt werden.

Zwar ist der erste Parameter immer anzugeben, hier wird er aber nicht mit einem *required* versehen, da unten ein Default auf den Hostobjektnamen gesetzt wird und damit immer gesetzt ist.

Erzeugt man nun einen Service an allen Linux-Maschinen, stellt man als Erstes fest, dass kein einziger Service ein OK meldet.

```
apply Service "hostname" {

  import "generic-service"

  check_command = "hostname"

  //assign where host.vars.os == "Linux"

  assign where host.name == NodeName

}
```

Codebeispiel 4.22: Service *hostname* beschränkt auf den Icinga-Server

Das kommt daher, dass in den Konfigurationen immer der FQDN als Objektname eingesetzt ist. Wer nun nicht das Check Command abändern möchte und einen Default hierfür setzen will, ergänzt das Template *linux-host* um *vars.hostname_fqdn = true*. Damit wird auf allen Linux-Systemen das Plugin mit `–fqdn` ausgeführt.

```
template Host "linux-host" {

  import "generic-host"

  vars.os = "Linux"

  vars.hostname_fqdn = true

}
```

Codebeispiel 4.23: FQDN ermitteln in *zones.d/templates.conf*

Als Nächstes stellt man fest, dass alle Hosts bis auf den Icinga-Server *fornax* selbst weiterhin ein WARNING melden. Das ist nicht verwunderlich, da es sich hierbei um ein lokal aufzurufendes Plugin handelt und für alle Services *hostname* das Plugin auf dem Server ausgeführt wird. Auf diesen Umstand bzw. wie das eigentliche Ziel zu erreichen ist, wird in Kapitel 5 ab Seite 83 eingegangen.

Eine Vielzahl von Check Commands befinden sich in der ITL und auch einige in Anhang F. Makros werden hingegen schon direkt im Anschluss näher behandelt, wenn es um die Ermittlung, die sogenannte Substitution, ihres Werts geht.

4.5 Makros und deren Substitution

Makros werden in $-Zeichen eingeschlossen. Es handelt sich bei einem Makro um eine Custom Variable oder Attribut gleichen Namens, dessen Werte aus den übergelagerten Objekten oder dem eigenen entnommen werden.

An einem Hostcheck sind je ein Objekt vom Typ *CheckCommand* und *Host* miteinander verknüpft. Hingegen ist bei einem Servicecheck zusätzlich noch ein Serviceobjekt beteiligt.

Ein Serviceobjekt *ssh* wird über eine *apply*-Regel bei Start von Icinga 2 für den Host *NodeName* erzeugt. Der Service ist wiederum über den Wert des Attributs *check_command* mit dem Check Command *ssh* verbunden. Das Check

Command sorgt für die Ausführung des Plugins `check_ssh`. Der Aufruf erfolgt nur mit der Option `-p`, wenn die Auswertung des Makros *ssh_port* einen definierten Wert ergibt.

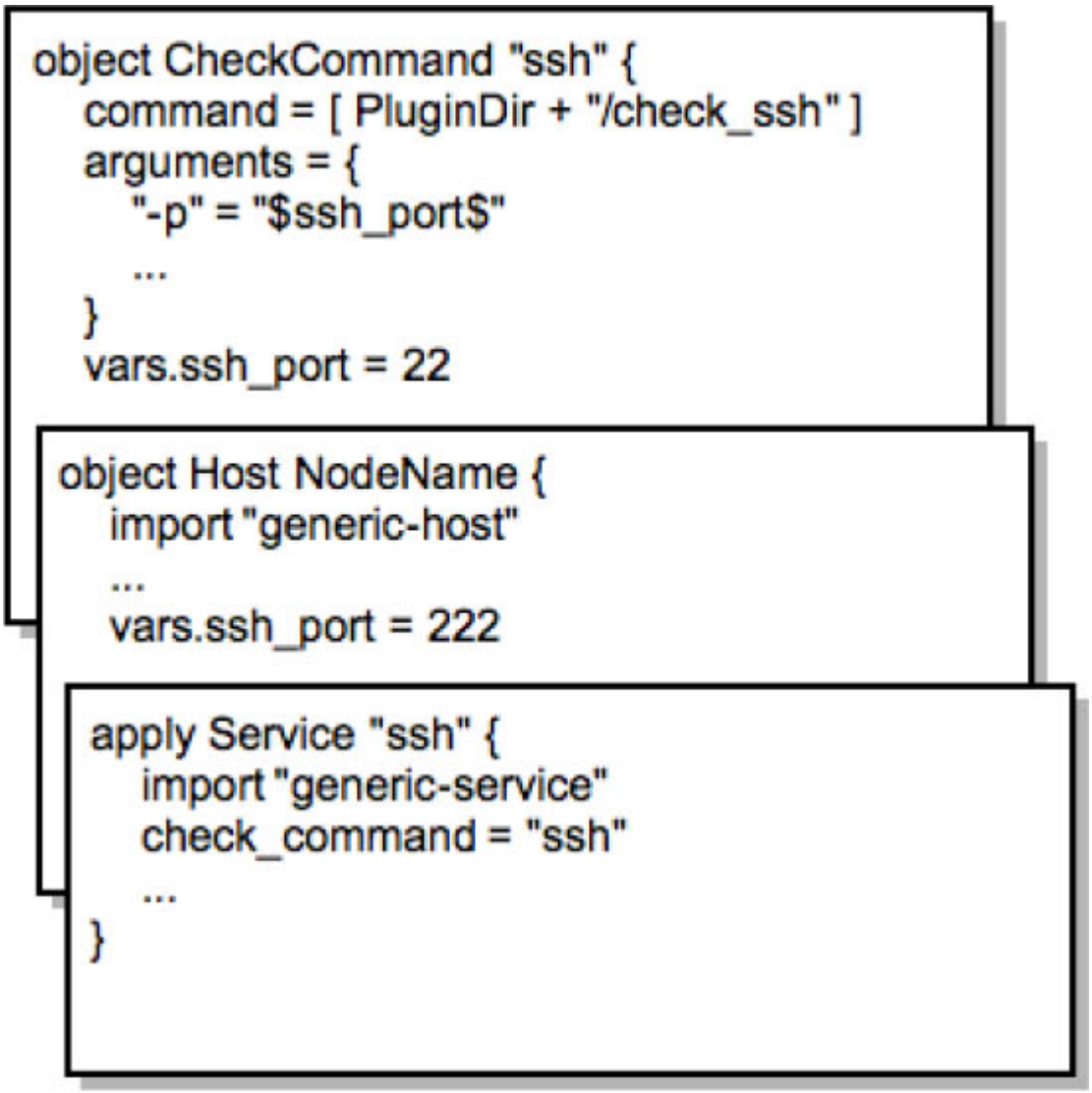

Abbildung 4-2: Makrosubstitution bei Hostattributen

Ist also ein Attribut bzw. eine Custom Variable mit entsprechenden Namen in einem der drei beteiligten Objekte gesetzt, wird das Plugin mit der Option aufgerufen. Ergibt die Auswertung jedoch, dass es mehrere Möglichkeiten gibt wie in Abbildung 4-2 im Hostobjekt mit *ssh_port = 222* und im Check Command mit *ssh_port = 22*, stellt sich die Frage nach der Wertigkeit. Welcher Port soll am Host also überwacht werden? Es wird der Port 222 geprüft, denn die Makrosubstitution räumt Attributen bzw. Custom Variables am Host eine höhere Wertigkeit ein. Im Check Command wird demnach lediglich ein Default gesetzt, der am Host überschrieben werden kann. Dieses Verhalten ist ganz klar an dem Ziel ausgerichtet, die Konfiguration, wie etwas zu überwachen ist, im Hostobjekt zu tätigen.

Käme noch die Custom Variable im Serviceobjekt hinzu, würde dieses den Wert im Host »überschreiben«. Damit ist die Reihenfolge bei der Makrosubstitution vom schwächsten zum stärksten Beteiligten wie folgt: Check Command, Host und Service. Was erst einmal verblüfft, sich aber am nachfolgenden Beispiel erklärt.

Betrachtet man nun das Beispiel in Abbildung 4-3, bei dem Serviceobjekte mittels einer *for*-Schleife erzeugt werden, wie es im vorhergehenden Abschnitt gezeigt wurde, wird jedem Serviceobjekt in der Variablen *config* ein Dictionary übergeben.

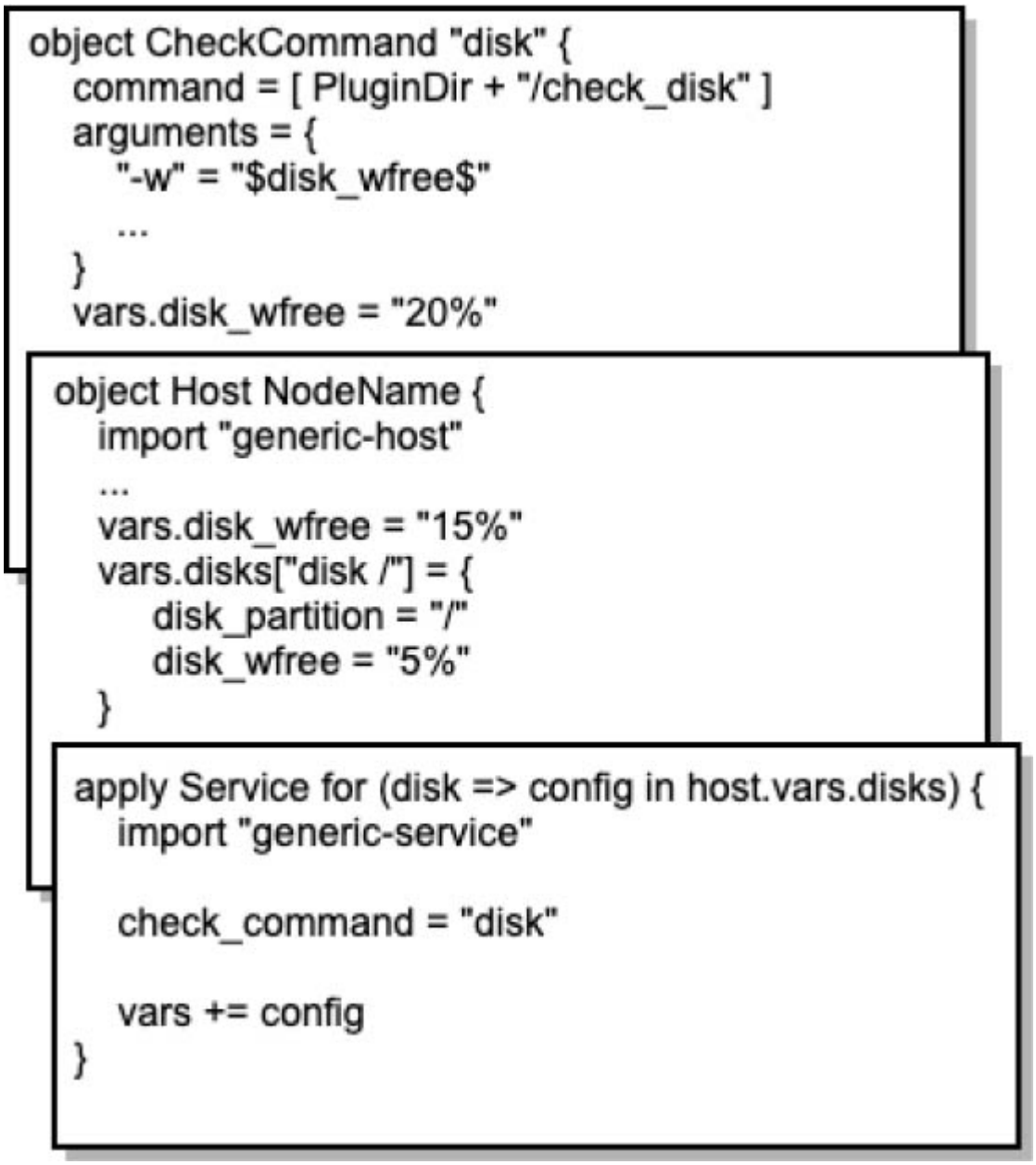

Abbildung 4-3: Makrosubstitution bei Serviceattributen

Der Inhalt besteht aus einfachen Key-Value-Paaren, bei denen der Schlüssel keine Custom Variable ist, denn diese sind ausschließlich als Schlüssel direkt unterhalb von *vars* definiert. Um nun aber unterschiedlich parametrisierte Plugin-Aufrufe für die verschiedenen Dateisysteme zu bekommen, wird ein Kunstgriff angewendet. Das Dictionary mit den eigentlichen Konfigurationsparametern in *config* wird auf Serviceebene mittels Merge »+=« dem Dictionary *vars* hinzugefügt. Damit werden aus diesen Einträgen Custom Variables in den einzeln erzeugten Services.

> Die Reihenfolge bei der Makrosubstitution vom schwächsten zum stärksten Beteiligten ist: Check Command, Host und Service.

Die aus Custom Variables resultierenden Optionen beim Plugin-Aufruf können damit z. B. für Dateisysteme unterschiedlich gestaltet werden. Zusätzlich ist es damit auch möglich, nicht nur generelle Standardwerte zu setzen, die systemweite Gültigkeit haben, sondern ebenfalls solche, die nur für genau einen Host gelten.

```
object Host NodeName {

  …

  vars.disk_wfree = "15%"

  vars.disks["disk /"] = {

    disk_partition = "/"

    disk_wfree = "5%"

  }

  vars.disks["disk /boot"] = {

    disk_partition = "/boot"

  }
```

```
}
```

Codebeispiel 4.24: Überschreiben von Standardschwellenwerten

Somit gilt ein Schwellenwert von 5 für WARNING im Codebeispiel 4.24 beim Dateisystem »disk /«, weil dieser Wert in eine Service-Custom-Variable überführt wird. Beim Dateisystem »disk /boot« sind es 15, da dieser Wert bei der Makrosubstitution nicht im Service gefunden wird, aber im zugehörigen Host und damit der Default aus dem Check Command »überschrieben« wird.

4.6 Timeperiods

Mit *timeperiods* werden Zeitfenster definiert, in denen Icinga 2 gewisse Aktionen ausführen darf. Im Umkehrschluss gilt, diese Aktionen werden außerhalb des definierten Zeitfensters nicht ausgeführt. In der Datei `conf.d/timeperiods.conf` stehen nach der Paketinstallation die drei Beispiele *24x7, 9to5* und *never* zur Verfügung. Zufällig entsprechen diese Definitionen genau den Anforderungen des Gebrauchtwarencenters und die Datei `timeperiods.conf` kann nach `zones.d/main` übernommen werden. Bei oder nach einem Kopieren muss auf die Berechtigung geachtet werden, siehe Abschnitt »Vorbereitungen« in Kapitel 4 ab Seite 54.

Die Periode *9to5* definiert Zeitfenster für die Werktage Montag bis Freitag und jeweils von 9 bis 17 Uhr. Das Zeitfenster des Timeperiod-Objekts *24x7* gilt rund um die Uhr, *never* deckt hingegen das andere Extremum ab, nämlich nie. Dem Attribut *ranges* wird ein Dictionary zugewiesen, die Schlüssel bestehen aus Wochentagen und die zugewiesenen Werte grenzen die Uhrzeiten ein. Als Schlüssel kann alternativ auch ein Datum verwendet werden. Die unterschiedlichen Notationen sind im nachfolgenden Beispiel enthalten und in dieser Form benutzbar.

```
object TimePeriod "9to5" {

  import "legacy-timeperiod"

  display_name = "Icinga 9to5 TimePeriod"
```

```
  ranges = {
    "monday" = "09:00-17:00"
    "tuesday" = "09:00-17:00"
    "wednesday" = "09:00-17:00"
    "thursday" = "09:00-17:00"
    "friday" = "09:00-17:00"
  }
}
```

Codebeispiel 4.25: Zeitfenster *9to5* in *zones.d/main/timeperiods.conf*

Als Zeitfenster lassen sich auch kommaseparierte Listen von Uhrzeiten benutzen. So werden am 21. Oktober 2022 nur die Zeitspannen zwischen 04:29 und 12:00 Uhr sowie zwischen 15:00 bis 24:00 berücksichtigt. Der 25. Dezember jeden Jahres ist als ganzer Tag in der Timeperiod enthalten. Das letzte Beispiel nimmt jeden 3. Tag ab dem 2. Montag im Februar bis einschließlich den 8. November in die Zeitspanne mit auf.

```
ranges = {
  "2022-10-21" = "04:29-12:00,15:00-24:00"
  "december 25" = "00:00-24:00"
  "monday 2 february - november 8 / 3" = "00:00-24:00"
}
```

Timeperiods werden sowohl in Host- wie auch in Servicechecks verwendet. Sie definieren die Zeitfenster, in denen die aktiven Checks ausgeführt werden. In außerhalb liegenden Zeiten wird dann keine Überprüfung vorgenommen. Weiterhin werden sie auch eingesetzt, um das Versenden von Benachrichtigungen zu regulieren. Für beide Fälle ist die voreingestellte Timeperiod *24x7*, die nicht einmal explizit definiert sein muss. Wird jedoch auf *24x7* konkret referenziert, muss so eine benannte Timeperiod in der Konfiguration vorhanden sein.

Mit dem gerade erworbenen Wissen über Timeperiods können nun auch die Checks für das Kassensystem und den Drucker des Gebrauchtwarencenters eingeschränkt werden. Ein Blick in die Onlinedokumentation[3] zu den Objekttypen Host und Service zeigt, dass das Attribut *check_period* für beide entsprechend auf die Timeperiod *9to5* zu setzen ist.

```
object Host "carina.gwc-jonas.local" {

  import "generic-host"

  address = "172.16.1.12"

  check_period = "9to5"

}
```

Codebeispiel 4.26: Check Period am Hostobjekt

Die Überlegung, dass auch die jeweils zum Host gehörigen Services in denselben Zeitperioden überwacht werden sollen, führt dazu, für Services generell dieselbe Timeperiod zu benutzen.

```
template Service "generic-service" {

  …

  check_period = host.check_period
```

```
}
```

Codebeispiel 4.27: Verwenden derselben Check Period am Service wie am Host

Ein Service existiert in Icinga nie ohne zugeordneten Host und so kann aus einem Service heraus, der aus einem *apply* erzeugt wird, immer auf Attribute und Custom Variables des Hosts zugegriffen werden.

4.7 Scheduled Downtimes

Downtimes sind ein sehr wichtiges Feature. Sie unterbinden den Versand von Benachrichtigungen in einem definierten Zeitfenster. Benachrichtigungen werden genauer in Kapitel 9 ab Seite 181 behandelt.

```
apply ScheduledDowntime "backup-downtime" to Service {

  author = "icingaadmin"

  comment = "Scheduled downtime for backup"

  ranges = {

    monday = service.vars.backup_downtime

    tuesday = service.vars.backup_downtime

    wednesday = service.vars.backup_downtime

    thursday = service.vars.backup_downtime

    friday = service.vars.backup_downtime

    saturday = service.vars.backup_downtime
```

```
    sunday = service.vars.backup_downtime

  }

  assign where service.vars.backup_downtime != ""

}
```

Codebeispiel 4.28: Beispiel einer wiederkehrenden Downtime für die Datensicherung

Neben dem Setzen von Downtimes über die grafische Benutzeroberfläche können regelmäßig wiederkehrende Downtimes in den Konfigurationsdateien definiert werden. Das Objekt *ScheduledDowntime* muss dabei immer dem Typ Service oder Host zugeordnet werden, da auch hier Host- und Serviceobjekte getrennt betrachtet werden. Mit dem Attribut *ranges*, das derselben Syntax unterliegt wie das gleichnamige Attribut vom Objekt *timeperiod* aus Abschnitt 4.6 auf Seite 73, werden Zeitspannen angegeben, in denen eine Downtime eingeplant werden soll.

Eine Datensicherung verursacht in der Regel einen erhöhten IO-Durchsatz und damit auf Unix-Systemen eine höhere Load als üblich. Mit obiger *apply*-Regel wird eine Downtime für jeden Service festgelegt, der die Custom Variable *backup_downtime* gesetzt hat. Angenommen, das Zeitfenster der Datensicherung ist von 2 bis 3 Uhr nachts, dann muss beim Service *vars.backup_downtime* der String *02:00-03:00* zugewiesen werden.

```
apply Service "load" {

  import "generic-service"

  check_command = "load"

  vars.backup_downtime = "02:00-03:00"

  assign where host.name == NodeName
```

```
}
```

Codebeispiel 4.29: Downtime setzen für den Service *load* zu Zeiten der Datensicherung

4.8 Debugging

Nachdem nun schon einige Objekttypen bekannt sind und sich die Definitionen von Objekten dieser Typen über mehrere Konfigurationsdateien verteilen können, wird hier mit `object list` ein hilfreiches Werkzeug vorgestellt, mit dem ein Überblick erhalten bleibt.

Durch den Aufruf von `icinga2 object list` auf der Konsole erhält man eine komplette Auflistung aller Objekte. Im Hintergrund wird bei diesem Aufruf die Datei */var/cache/icinga2/icinga2.debug* ausgewertet. Sie wird bei jeder Validierung der Konfiguration aktualisiert, also bei einem Start, Neustart oder Reload. Allerdings geschieht das auch bei einem expliziten Aufruf, die Konfiguration zu überprüfen, wie es mit `icinga2 daemon -C` per Hand erfolgen kann. Daraus folgt, es muss sich nicht um die aktuell benutzte Konfiguration handeln! Das heißt aber auch, man kann sich Änderungen anschauen, bevor sie »live« gehen.

Mit der Option `–type` kann das Ergebnis auf einen bestimmten Objekttyp eingeschränkt werden. Wird zusätzlich noch `–name` verwendet, ist die Anzeige auf ein Objekt eingeschränkt.

```
$ icinga2 object list –type host –name antlia.gwc-jonas.net

 % declared in '…/main/hosts/antlia.conf', lines 25:1-25:20

 * __name = "antlia.gwc-jonas.net"

 * action_url = ""

 * address = "172.16.2.12"

   % = modified in '…/main/hosts/antlia.conf', lines 30:3-
   30:23
```

```
  * address6 = "::1"

    % = modified in '…/main/hosts/antlia.conf', lines 31:3-
    31:18

  * check_command = "hostalive"

    % = modified in '…/main/templates.conf', lines 19:3-19:29

  * check_interval = 60

    % = modified in '…/main/templates.conf', lines 16:3-16:21

…
```

Beim Start oder auch bei einem erneuten Reload von Icinga 2 werden z. B. alle Template-Importe durchgeführt und damit aufgelöst. Das wirklich Schöne an der Objektliste ist, dass die Information gegeben wird, aus welcher Datei ein Attribut stammt.

Bei Icinga 2 existiert ein Service nie alleine, sondern ist immer einem Host zugeordnet, damit setzt sich der intern in Icinga verwendete eindeutig bezeichnende Name immer aus dem des Hosts und des Services zusammen. Beide werden mit einem »!« voneinander getrennt geschrieben, damit ist der Servicename in der Konsole zu quoten.

```
$ icinga2 object list –type service \

        –name 'antlia.gwc-jonas.net!ping4'

Object 'antlia.gwc-jonas.net!ping4' of type 'Service':

  % declared in '…/main/network.conf', lines 26:1-26:21

  * __name = "antlia.gwc-jonas.net!ping4"
```

```
  * action_url = ""

  * check_command = "ping4"

  …
```

Eine Timeperiod ist hingegen wiederum ein alleinstehendes Objekt und eine einzelne Periode wird mit ihrem Namen abgefragt. Eine Scheduled Downtime setzt sich aus dem Namen des Objekts, auf das sie sich bezieht, und als Erweiterung aus dem Namen des Hostobjekts und dem Unix-Zeitstempel des Eintrittzeitpunkts der Downtime zusammen. Zusätzlich wird noch eine Zahl als Index angehängt, falls mehrere Downtimes mit derselben Anfangszeit angelegt wurden.

Für eine Downtime zum Service *load* auf dem Host *fornax* sähe der eindeutig identifizierende Name dann wie folgt aus:

```
'fornax.gwc-jonas.local!load!fornax.gwc-jonas.local-15…-0'
```

Handelt es sich um eine Downtime zu einem Host, verkürzt sich der Name wieder auf zwei durch »!« getrennte Teile, da der Service entfällt.

Möchte man eine Konfigurationsänderung auf Syntax und Objektabhängigkeiten testen, erreicht man dies mit dem Kommandozeilenaufruf von `icinga2` mit dem Subkommando `daemon` und der Option `—validate` bzw. `-C` in der verkürzten Schreibweise.

Treten keine Fehler auf, zeigt der Aufruf der Validierung lediglich, wie viele Objekte von welchem Typ in der Konfiguration enthalten sind. Die zu diesem Zeitpunkt sehr hohe Anzahl von Check Commands rührt daher, dass diese in der ITL definiert sind und standardmäßig auch alle geladen werden.

```
$ icinga2 daemon —validate

information/cli: Icinga application loader (version: r2.13.2-
1)

information/cli: Loading configuration file(s).
```

```
information/ConfigItem: Committing config item(s).

information/ApiListener: My API identity: fornax.gwc-jonas…

information/ConfigItem: Instantiated 228 CheckCommands.

information/ConfigItem: Instantiated 1 IcingaApplication.

information/ConfigItem: Instantiated 3 Host.

information/ConfigItem: Instantiated 1 FileLogger.

information/ConfigItem: Instantiated 1 Comment.

information/ConfigItem: Instantiated 1 IcingaApplication.

information/ConfigItem: Instantiated 3 HostGroups.

information/ConfigItem: Instantiated 1 CheckerComponent.

information/ConfigItem: Instantiated 1 Zones.

information/ConfigItem: Instantiated 1 Endpoints.

information/ConfigItem: Instantiated 1 IdoMysqlConnection.

information/ConfigItem: Instantiated 1 ApiUser.

information/ConfigItem: Instantiated 1 ApiListener.

information/ConfigItem: Instantiated 253 CheckCommands.

information/ConfigItem: Instantiated 3 TimePeriods.
```

```
information/ConfigItem: Instantiated 14 Services.

information/ConfigItem: Instantiated 3 ServiceGroups.

…

information/cli: Finished validating the configuration
file(s).
```

Codebeispiel 4.30: Ausgabe der Konfigurationsvalidierung

Die Ausgabe zu aufgetretenen Fehlern ist sehr hilfreich. So informiert die folgende Ausgabe darüber, dass das Host-Template *generic-host* in der Konfiguration nicht gefunden werden konnte. Da jedoch das angegebene Hostobjekt auf dessen Verwendung besteht, ist die Konfiguration nicht korrekt und es wird mit einem Fehler abgebrochen.

```
critical/config: Error: Import … unknown …: 'generic-host'

Location: in …/zones.d/main/hosts/antlia.conf: 20:3-20:23

…/zones.d/main/hosts/antlia.conf(18): object Host NodeName {

…/zones.d/main/hosts/antlia.conf(19):

/* Import the default…

…/zones.d/main/hosts/antlia.conf(20): import "generic-host"

                                      ^^^^^^^^^^^^^^^^^^^^

…/zones.d/main/hosts/antlia.conf(21):

…/zones.d/main/hosts/antlia.conf(22):
```

```
/* Specify the address…
```

```
critical/config: 1 error
```

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z