How DetectRight Detects User Agents

High Level Description

By Chris Abbott

Abstract: This paper looks in high level detail at the detection strategies used by DetectRight to identify manufacturers, models and components. It compares DetectRight's "Tenacious Profile" approach to what we call the "Simple Match" approach taken by other systems: in this case we consider WURFL and DeviceAtlas.

Intended audience: users of the DetectRight standalone device detection libraries. WURFL and DeviceAtlas users who wish to see DetectRight's algorithms.

V1.0 – Initial draft

V1.01 – edited DeviceAtlas and WURFL Detection to add additional information, clarified small details.

Background

HTTP Headers

One of the main functions of any device detection system is to deal with "User-agents": unique text strings that are sent to web sites by mobile and non-mobile browsers.

"User-Agent" is one of many "HTTP Headers" that a website receives during a web access.

Sample headers:

```
User-Agent: Nokia3510i/1.0 (4.43)
Accept: text/html,audio/mid,image/gif
```

These "HTTP headers" convey extra information about the calling device and its journey through the Internet.

There are hundreds of headers to be seen in the wild, but most device databases only care about "User-Agent", which is the standard method for connecting a request to a data entry.

Unfortunately there are no binding standards affecting how User-Agent strings are used, what they should contain, nor rules about how they should be formatted or changed.

The result is that historically there have been over 2,000,000 different User-Agents. Local attempts to enforce consistency within bigger organisations have failed. So User-Agent format fragmentation is the biggest problem affecting detection today.

User-Agent detection and analysis

There are two main use cases for useragent strings:

1) To identify what's accessing the site
2) To identify what additional components are running, and convey other "preference" information

Most device databases concentrate on step 1: using a useragent to identify a device, and either ignore step 2 or deal with it manually.

Identification

The process of convert a useragent string to some kind of device identifier which can then be used to look up a set of static properties

Here's an example of a simple user agent:

```
Nokia3510i/1.0 (4.43)
```

Here's an example of a Mozilla user agent from a current Android device:

```
Mozilla/5.0 (Linux; U; Android 2.2.2; en-gb; LG-P500/v10q Build/FRG83) AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0 Mobile Safari/533.1 MMS/LG-Android-MMS-V1.0/1.2
```

Anyone can see that these User-Agents correspond to a Nokia 3510i and an LG P500 (which is an LG Optimus One).

But a detection system needs algorithms to work this out.

Once a system knows what device it's looking at, it can build its profile.

"Ideal" Detection strategies

There are two scenarios for ideal detection. Neither is possible in the real-world. However, a remarkably high number of systems have been designed as if it was!

> *Match user-agent and device with huge 100% accurate table*
>
> *Have no database, and use intelligent string matching to recognise all useragents*

Match user-agent and device with huge 100% accurate table

Why won't this work as a pure strategy?

1) There's no public catalogue of published user-agents: you have to create your own catalogue from web traffic. You will only ever see some of them.
2) New user-Agents are always being created by manufacturers, and even users. The database will always be out of date.
3) There are over 1,000,000 useragent records in DetectRight alone. The dataset is simply too big for distributing (any gigabytes), and too large to search in real-time without proper relational database technology.

Have no useragent table, and use intelligent string matching to recognise all useragents

Why won't this work as a pure strategy?

1) Useragents are simply too varied and messy to have simple rules governing them
2) Someone has to work out all the rules, past, present and future. It's not something that can be automated
3) The ruleset has to be created from examining real user-agents: any consistent rules for analysing user-agents are made useless by the number of exceptions
4) If you're not careful, the list of expressions to look for just becomes a list of user-agents again (i.e. a database, but an inefficient one).
5) New rules always need to be created to keep up with new formats or developments, without disrupting old rules.

Conclusion

Any detection strategy has to deal with the following issues:

- Available data will always be incomplete. There will always be missing and new user-agents hitting the system.
- New devices will hit the wild before your system has meaningful data attached to them: these are real customers who may well be lost.
- Even if you had all user-agents ever seen, it would be impractical to deploy them for a real-time system.
- There is no unifying set of rules that will help in detecting unseen user-agents sight unseen

How a detection system deals with these problems determines how good it is at its core mission: producing a data profile that matches the customer's device as closely as possible, so it can be served better data.

DetectRight Detection

DetectRight has an "entity" based approach.

An Entity is any conceptual object, such as a Mobile Phone (DetectRight uses the generic term "Device" for this), a Tablet, a Browser, an OS, a JVM, a CPU, a Chipset, a GPS, or any other "thing".

In DetectRight's world, an Entity can also contain other entities. For instance, a Device has an OS, a Browser, a CPU, a broadband chipset, etc. This contains information is useful because it allows sensible "Device Families".

This mirrors the way that companies themselves think of handsets: they start with a manifest of components, and then apply device-specific tweaks to the end result to achieve a particular marketing aim.

DetectRight's approach to detection is to build an inventory of components and their versions ("an EntitySigCollection"), then create a master data tree ("PropertyCollectionTree").

1) Clean user-agent (this includes replacing language strings after it has recorded the information)
2) Check user-agent against manual overrides to check for attached components.
3) Run user-agent against over 800 tests to build a "manifest" of detected components, and their version numbers. This creates an inventory of components.
4) Find the component that represents what the device actually is (e.g. Device, Nokia 3510i)
5) Add any components that ship with the device but which weren't detected by step (3)
6) Build a data tree representing the inventory
7) Interrogate tree to generate device profile

Example: Detecting a Simple Useragent

```
User-Agent: Nokia3510i/1.0 (04.43) (via Google Translate)
```

Step 1: Clean the User-Agent

```
Result: Nokia3510i/1.0
```

User-agents tend to acquire extra unwanted information if they go through other processes or gateways. This is a major source of problems for device detection systems.

Step 2: Look it up

```
Result: {nothing}
```

Why does DetectRight not match this? The device profile on the website shows this user-agent so we know it's in the main system.

The answer is: *because DetectRight's detection database is generated with only entries that fail Step 3 (user-agent tests).*

When Detectright generates a database for a user, it checks every useragent in the system by running Step 3 on it. If the result from detection is different to the existing mapping, it keeps the useragent. If it's the same, it discards it.

Step 3: Run tests on it

The tests work purely on the visible data in the user-agent. In this case, DetectRight would only find the following:

```
Result:Device:Nokia:3510i::1.0
```

(i.e. the Device Nokia 3510i, at version 1.0)

Step 4: Work out what "it" actually is

For all the components detected in Step 3, DetectRight looks for the one which is most likely to "be" the device by looking at the "entity type" (e.g. "Device", "Tablet", "Fridge").

In this case, there's one entity which is a "Device". So the choice is a no-brainer. This "is" a Nokia 3510i.

Once we know what the "Device" is, we have a proper Device Id (a 32 character string), which uniquely identifies it in the database. So we can find out what else the database knows about it.

NOTE: if DetectRight has not successfully found a device (for instance, if there isn't one, or it knows the device isn't in the database), two things could happen here, depending on settings:

i) Process stops, error is generated
ii) Process continues and adaptive profile is built from the other components

Step 5: Fill in the gaps

There are some things we know about the Nokia 3510i that aren't visible in the user-agent.

For instance, in its default state it contains the following components/Entities

OS:Nokia:Nokia OS (i.e. Nokia proprietary OS, which underpins Nokia Series 40)

Developer Platform:Nokia:Series 40::1.0 (Nokia Series 40, first edition)

Browser:Browser:Nokia

Each of these components comes with its data tree.

Where a component hasn't been already filled, DetectRight adds these into the inventory. For instance, if the device was running another browser, the data from Browser:Browser:Nokia wouldn't be added, since a device can't be running two browsers in the same website access.

Step 6: Generate a tree ("PropertyCollectionTree")

Once we have the list of components and versions, DetectRight merge all the trees into a master tree. This can then be interactively asked questions to get data about the device.

Step 7: Generate profile for requested schema

WURFL and DeviceAtlas have their own set of datapoints ("schema") that they output. These represent single leaves in a much bigger possible tree of information.

DetectRight takes the list of required datapoints and queries the data tree with each one for a profile value.

Footnote: For instance, the data "width in pixels of the primary display screen?" corresponds to "resolution_width" in WURFL and "screenWidth" in DeviceAtlas. DetectRight gets that information from the tree like this: Display//0//dimension=size{value:*;units:pixels;arg:0}. You don't need to know this to use DetectRight!

Step 8: Override the profile

DetectRight's web user interface allows users to attach their own override data at device/Entity level. This data is superimposed onto the profile from step (7) just before output. This guarantees that override data will always appear in the final data.

Step 9: Return the profile

WURFL Detection

WURFL's strategy is based on what it terms "device ids". However, a more accurate name would be "useragent id", since the word "Device" is more commonly used to describe a physical piece of equipment. These Ids aren't guaranteed to remain stable over time, and have in the past often been subject to "cleaning and improvement".

WURFL's user-agent strategy:

1) Clean user-agent
2) Look it up
3) Run tests on it to get a Device Id
4) Build profile from WURFL tree based on that Id
5) Return profile

Example: Detecting a Simple Useragent

```
User-Agent: Nokia3510i/1.0 (04.43) (via Google Translate)
```

Step 1: Clean the User-Agent

```
Result: Nokia3510i/1.0 (04.43) (via Google Translate)
```

WURFL's cleanup is much more rudimentary than DetectRight's.

Step 2: Look it up

```
Result: nothing
```

There are millions of user-agents. WURFL has only tens of thousands. So "recovery match" will be much more common than "exact match", especially across new traffic.

This means that WURFL's overall accuracy is highly dependent on its "recovery match" features. See "Detection Failure Strategies".

Step 3: Run tests on it to match it (conclusive heuristic match)

```
Input: Nokia3510i/1.0 (04.43) (via Google Translate)
Result: Nokia3510i/1.0 (04.01) Profile/MIDP-1.0 Configuration/CLDC-1.0
(device_id = nokia_3510i_ver1_sub0401)
```

WURFL has picked what it thinks is the nearest user-agent to ours, and chosen a device ID. In this case, it got it right by reducing the useragent repeatedly until it got something it recognised (in this case, basing the detection on the common pattern "`Nokia3510i/1.0 (4.`".

As a test, what happens if we change the initial "N" to an "n"? WURFL fails the detection completely, indicating the WURFL is very sensitive to spacing, and capitalisation: both things which can get changed in the chaotic ecosystem!

A "recovery match" checks for certain strings in the UserAgent and hardcodes device Ids to them if the conclusive heuristic has failed.

NOTE: WURFL "takes the requester's user agent and puts it through a filter to determine which UserAgentMatcher to use on it. Each UserAgentMatcher is specifically designed to best match the device from a group of similar devices using Reduction in String and/or the Levenshtein Distance algorithm."
(source: http://www.tera-wurfl.com/wiki/index.php/Main_Page)

Both main comparison methods are based purely on string similarity. This is a major flaw in the WURFL system (and DeviceAtlas too): the key recognition algorithm in the system treats user-agent strings like any other string, and takes no account of the context-specific semantic content inherent in a user-agent. This means that WURFL and DeviceAtlas are sensitive to spacing and order issues. DetectRight is not.

Step 4: Build a profile using the device Id

Next, WURFL builds a profile by journeying backwards through a tree from the device Id `"nokia_3510i_ver1_sub0401"` filling in data as it goes.

Each user-agent in the WURFL XML dataset is given a matching ID. A user-agent in WURFL acts as a sample. It might be matched to other user-agents by WURFL. Here's the the Nokia 3510i entry we just found:

```
<device id="nokia_3510i_ver1_sub0401" user_agent="Nokia3510i/1.0 (04.01) Profile/MIDP-1.0 Configuration/CLDC-1.0"
fall_back="nokia_3510i_ver1_sub">
```

The data from that entry is stored to start the profile off. Then WURFL gets the "fallback" entry. It looks like this:

```
<device id="nokia_3510i_ver1_sub" user_agent="Nokia3510i/1.0" fall_back="nokia_3510i_ver1">
```

So far we haven't picked up any data except "data rate", but that's about to change.

In WURFL, most data is stored at "root device" level (actual_device_root = "true"), which is the next fallback:

```
<device id="nokia_3510i_ver1" user_agent="Nokia3510i" fall_back="nokia_generic_series40" actual_device_root="true">
```

From here, WURFL continues to fall back through more and more generic devices, adding data as it goes:

```
nokia_generic_series40
nokia_generic_series30
nokia_generic_series20
nokia_generic
generic
```

Footnote: The last step adds a lot of the data from the "generic" ID. DetectRight does not pad out profiles like this, since we believe that generic data gives a false impression of confidence in the data. There's a big difference between "false" and "we don't know", which leads to many users being mishandled on content sites.

Important note:

WURFL does not use the user-agent for any other purpose than matching to a device Id. That means it generally ignores changes to the user-agent that may signal even a major change in functionality, such as OS upgrades: the exception to this is if the change has been specifically dealt with with override datapoints for a specific user-agent: how this change would generalise is up to the LD algorithm, which doesn't assign this change more importance than anything else.

Footnote: WURFL has recently acquired a "BrowserID" field, and proposals are apparently afoot to separate out browser data from device data and providing a link from a node to a browser heirarchy. This seems like a rudimentary attempt to begin to address the issues that are solved by DetectRight's component model, but lacks the conflict resolution mechanism, and also requires a lot of manual data per user-agent for useragents representing a switched browser. For instance, if a device has browser characteristics currently, those implicitly belong to its shipped browser: which means that the profile for the device takes priority over the browser data, to ensure that device specific browser data is not lost. However, if the browser has been swapped, then the browser data takes priority over the device browser data. Which means that the combine operation should work differently. Essentially although this BrowserID approach fixes some of the problems with data maintenance and begins to address componentisation, it is itself constrained by the fallback hierarchy and a lack of mechanism for dealing with conflicts in real-time: both of which are problems DetectRight has engineered away.

DeviceAtlas Detection

DeviceAtlas's approach to the problem is in a way opposite to WURFL's, in that instead of reducing user-agents character by character to get a match, it builds up the match string character by character, adding data as it goes. As it adds characters, it follows nodes in a tree.

Step 1: Clean the User-Agent

`Result: Nokia3510i/1.0 (04.43) (via Google Translate)`

DeviceAtlas's initial cleanup is much more rudimentary than DetectRight's and even WURFL's. Unusually, DeviceAtlas does trigger the removal of parts of a useragent at certain tree nodes: this was introduced in API 1.4 to make Android detection less disastrous (prior to this, it was choosing the Android model based on the language string). The "improved" cleaning process unfortunately erases such information as proper OS or Developer Platform version, so DeviceAtlas's coping strategies have consequences for profile accuracy.

The cleaning process also doesn't appear to adjust for case sensitivity. This is a major infrastructural error.

Step 2: Look it up

DeviceAtlas doesn't have "recovery heuristics". It collects data until it runs out of tree nodes, then delivers whatever profile it has picked up.

This mechanism relies heavily on the useragent being unaltered: even spacing and case-sensitivity matter. Why is this a problem? Because the ecosystem is chaotic! User-agents are being altered all the time: things being added to the start, case changes, spaces being added, etc.

Let's feed DeviceAtlas the unaltered useragent. It detects a Nokia 3510i, as expected, based on "Nokia3510i"

`Result: Nokia3510i/1.0 (04.43) (via Google Translate)`

But what happens if we change the useragent to have a lower-case "N"? It detects a Nokia 3650. DeviceAtlas's tree thinks that any Nokia string beginning "nokia3" (lowercase) is the Nokia 3650!

Result: `nokia3510i/1.0 (04.43) (via Google Translate)`

But why? It's because the data tree itself was created with case-sensitive nodes. Once the process has gone down the "n" node instead of the "N" node, it's doomed.

What happens if a space gets introduced? Well, all it knows now is that it's a generic Nokia of some kind.

`Result: Nokia 3510i/1.0 (04.43) (via Google Translate)`

In this scenario, the "recovery match" is the data picked up while travelling to the end of the tree. DeviceAtlas's design depends on the branches of the tree being as short as possible, so user-agents are truncated in their dataset wherever possible. DeviceAtlas does not attempt to find a "suitable device", though it is prone to inappropriately using partial model matches. Analytics based on this would be deeply unreliable.

Step 3: Return profile

Whatever it picks up so far, it returns.

Footnote: The DeviceAtlas tree itself falls victim to the "complete problem space delusion": the compilation process assumes that the useragents in DeviceAtlas comprise the whole Universe of known user agents, which means that the system often makes unwarranted assumptions about devices based in user-agent fragments which are unjustifiably short: for instance, assuming that all user-agent strings starting with "nokia3" belong to the Nokia 3650. This is not a safe real-world assumption, and can easily be broken. It's certainly true for the DeviceAtlas dataset, but it's not a justifiable assumption for the problem space generally (the problem here is made worse by the case sensitivity). DeviceAtlas has only 1/3 or less of total devices, and a very small percentage of useragents in the real world. Essentially they are relying on people's update frequency to mitigate this flaw in their theoretical approach to detection. It also puts possibly unwarranted faith in the device update workflow at source.

What happens with new Devices?

New devices come along all the time, driven by newly enthusiastic users with money to burn.

Unfortunately it's these same users who are mostly likely to be disappointed by the mobile web.

We at DetectRight believe this is one reason why browser manufacturers have chosen to implement "cloaking schemes" to hide their device's identity ("Desktop view", for instance): because the general standard of detection on the web for new devices is disappointing: new devices are likely to be served content meant for phones from 2001 or 2002: the result of detection misses in current systems, or even worse, simple regexp scripts.

Let's create a new device!

Oh dear, HTC have just released the "HTC SuperDuper"! The useragent is this:

```
Mozilla/5.0 (Linux; U; Android 2.3.3; en-us; SuperDuper Build/FRF91) AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0 Mobile Safari/533.1
```

At this point we know that each database does not have any specific information on the device, such as screen-size. We also know that as an Android device, the user has very powerful multimedia capabilities: so even if we don't know the model number, we should be able to serve multimedia, and know it's a mobile device. Let's say we're serving a beautiful MP4 video, and we need the device detection to tell us whether to serve it. Any system worth the name should know that Android 2.3 devices can do H264.

DeviceAtlas: FAIL.

```
Mozilla/5.0 (Linux; U; AndroidS2.3.3; en-us; SuperDuper Build/FRF91) AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0 Mobile Safari/533.1
```

The only profile data acquired is that it's a mobile device, since it knows of no Android models beginning with "S", apparently. No other data at all is delivered (which is quite odd, since you'd think they'd attach some generic Android data to the "d" node of "Android". In this case, you've got one disappointed user, who might well be served a very basic page.

WURFL: Semi-pass

```
Android SDK  - DO_NOT_MATCH_GENERIC_ANDROID
```

In this case we've got an SDK with "Android version 0.5", thanks to the presence of the string "Android". Screensize of 240x320, unfortunately does not tally with the official Android specs for Gingerbread, which has a much bigger minimum screensize. MP4 support is present though, since it's present in all versions of Android. If the detected capability relied on OS version, it would be much more of a problem.

DetectRight: Pass

Not only does DetectRight recognise the device as "Android SuperDuper" (the best it can do without the manufacturer name in the user-agent), but it adds in all the properties of Android 2.3 OS and the Android Webkit browser: and that includes a healthy screensize of 760x1280. That's the difference between a user getting a postage stamp, and a proper screen. If the Android name was "HTCSuperDuper", then it would recognise it as an HTC SuperDuper, but otherwise behave the same.