The state of RapidSMS

May, 2010

Summary

RapidSMS is a powerful software, successfully used to build SMS based services delivering crucial health data in real time for more than xxxx patients in country X, country Y and Country Z.

Despite its success story, **RapidSMS suffers from several major deficiencies that may endanger the current projects durability and the realization of the already numerous incubating ones**:

- It follows no standards what so ever and even erases existing ones;
- It has a hardly maintainable code base;
- It does not allow test automation;
- It has almost no documentation;
- It is already fragmenting in several incompatible unofficial versions.

First, it means that **developing with RapidSMS require more time and money that it should be**. More importantly, it means that, as project using it will multiply, the number of person able to deal with its complexity will not follow. **Hiring will become a bottleneck** and training locals to independently maintain the currently running systems is not possible in these conditions.

Without anybody to be able to deal with the technology, the projects will little by little decline. Instead of trying to train more and more developers and fill wholes in the design on the way, a pragmatic solution is to rewrite it so it becomes easier to use, more reliable and cost effective.

In the following report, we propose to invest in a sane rewrite now, while the number of projects using it -and so the technical debt- is still low. Gathering the experience of several professional developpers, we identified the following key point of what could RapidSMS becomes:

- Turning RapidSMS into a pluggable Django app, easy to install, maintain and deploy
- Make RapidSMS use Django tools such as urlconf-like routing, views, templates and middlewares.
- Create a documentation and push conventions/best practices.
- Provide sane and useful default: “there should one and only one obvious way to do it”.
- Improve debugging tools.

Index

RapidSMS works

RapidSMS is today the fastest and easiest way to provide an efficient and reliable SMS service at a reasonable cost, ensuring critical informations to circulate at a fast path in challenging environments. It already powers several charity projects that matters, including:

The tool has especially the tremendous ability to perform well with cheap hardware anybody can buy, and make it work with virtually any mobile phone providers without any particular set up. Because of its open license and code, NGOs can use it and adapt it to their needs, in order to create custom text message information systems that make a big difference in areas where communication is vital but yet bound to very poor national equipments.

Therefor, the following report is not, by any mean, an effort to discredit RapidSMS nor its creators, but rather of review of the main enhancements it would require to be used at its full potential.

RapidSMS deficiencies

It is because RapidSMS is such a useful tool and because several other majors projects are starting to emerge around it that it becomes important to invest in it, by fixing major flows that could cost a lot on the long run.

No matter how useful RapidSMS is, it suffers from very important deficiencies. At best, they slow down the expansion of it's adoption and the evolution of any project using it. At worst, it endangers the quality of the results and rise dramatically the cost of development then maintenance of the software part.

Still, it remains the best solution available, but lacking of better alternatives should not comfort anyone in the status quo. At the contrary, it should be urgent to move forward before getting stuck at a point where one will spend as much time and money with compensating the problems instead of working on what matters.

You will find here a synthesis of feedbacks from several professional developers.

No integration with any standard tools

RapidSMS as been built at the top of a technology named Django, a famous flexible and powerful Web framework with a strong open source community to support it. What sounds like a good idea was unfortunately poorly implemented, and instead of been integrated in Django like any other application of the Django ecosystem, it replaces arbitrarily several parts of it.

As a consequence, it is very hard to make it work with any standard tool, plug-in, application or library that would have been Django friendly. Any improvement made to Django itself, such as security fixes, is almost impossible to reintegrate in RapidSMS which is now a static tools, very uneasy to make evolve or play with other tools. Integration and deployment phases gain in weight and complexity, while surrounded with incertitudes. Using handcrafted custom scripts is the norm.

For the projects, it means a lesser final quality and a longer development time.

But worst, it means that you just can't take any Python/Django developer, among their vivid community of quality programmers, and ask them to work on RapidSMS, assuming they will find their way in. It systematically implies training them, making them working with most RapidSMS experienced programmers, and add time for them to analyze RapidSMS itself before using it, instead of spending these resources on the project.

Not only it is a waste of time and money, but its a consumer of one particular resource much harder to renew : good developers. In order to work on a RapidSMS project, you must find somebody with sufficient skills to do so. These task is already not easy for a normal job and even less if the mission is based in a third world country. But it becomes tricky with a technology that is itself a challenge to use, and not in the pleasant way. Eventually, projects will be queuing, waiting for the appropriate human resource to be freed so it can work on it and staff competence and availability will become a major bottleneck.

No tests

There is no automated way to test if the RapidSMS code work on your current machine, which means there is no way to know is any change you make to it has affected in anyway to rest of the software.

More importantly, standard Python and Django modules for testing simply don't work anymore under RapidSMS. Running `./rapidsms test` purely crashes the entire system. Not only can't you test the RapidSMS code and must trust it, but you can't test your own code to ensure it works.

Developing with RapidSMS requires painful, error prone and slow manual tests, a work flow that is not really a best practice when dealing with sensitive data such as medical results where reliability is a priority.

The code base cannot be maintain

The code base has been written to work, but not to evolve. RapidSMS coders were obviously competent programmers but newcomers to Django, and achieved their goal by any mean they could think of. They are themselves well aware of it, as you can read them in this extract of their code. Pay attention to their comments, here in light blue, precessed by a '#':

```
45      # A NEW KIND OF LUNACY: inspect the stack to find out
46      # which rapidsms app this function is being called from
47      tb = traceback.extract_stack(limit=2)
48      sep = os.sep
49      if sep == '\\':
50          # if windows, the file separator itself needs to be
51          # escaped again
52          sep = "\\\\"
53      m = re.match(r'^.+%s(.+?)%sviews\.py$' % (sep, sep), tb[-2][0])
54      if m is not None:
55          app_type = m.group(1)
56
57          # since we're fetching the app conf, add it to the
58          # template dict. it wouldn't be a very good idea to
59          # use it, but sometimes, when time is short...
60          rs_dict["app_conf"] = settings.RAPIDSMS_APPS[app_type]
61
```

In fact, any changes to RapidSMS would require a cautious time consuming analysis, followed by slow and hazardous modifications, to potentiality ends up with unexpected bugs there is no way to detect before the system crashes.

RapidSMS is almost set in stone. Bug fixes, security patches and new features are not something you can casually plan, if you can plan it at all.

But we do add patches, nevertheless, because like any software RapidSMS can not answer to every needs and contains bugs. It's natural, any program does. Trouble is, nobody is quite sure of the consequences of theses patches because, again, there is no way to test them.

Furthermore, every team tends to tweak RapidSMS on the fly, as they can and build little buy little their own incompatible yet more stable versions. Which leads to the next issue.

Fragmentation

Since everybody needs RapidSMS to evolve, they make it so, but in an anarchical informal way. Patches are exchanges by emails, several teams now work with hybrids of RapidSMS, completely incompatible with each others with more of less features and bug fixes. Each team want only work with it's version because it spent time on it and understand it.

Work that have been done is redone. RapidSMS itself doesn't benefit at all for the various improvements because nothing is coordinated, and the main software itself evolutes slowly.

Therefor, working on one RapidSMS then another can imply substantial differences, with again consequences on quality and costs.

No documentation

There is now no way to learn RapidSMS by yourself for most of programmers. The most experimented can dig in the code, and spend days in trials and erros to finally get their hands on the basic. The others need a documentation. For both, this is less than an optimal situation.

No documentation means that for now, learning RapidSMS require an experienced RapidSMS programmer to be dedicated to teach the new one needed to integrate the team. But soon, the newcomer realizes that “experienced” RapidSMS programmers are still at the stage of exploration themselves.

No documentation means no common practices again. If one takes a RapidSMS developer form another team, there is a good chance that he would not performs the job the same way. Even more disturbing, a beginner can been explained how to do the same thing in two different maners by two members of its team.

You can not really rely on other application code either to show the way, because none of they do the job quite the same way.

For deployment of course, this has another consequence: there is no such thing as a list of supported systems and equipments. Some would even work but only after altering the RapidSMS code, since some settings are hard coded. Hopefully, the require hardware price is low, which allow trials and errors, and eventually each team use the same equipment that is known to work.

Of course, every team has its own internal documentation, but they are not shared, not that it would be surely helpful since they tend to be specific to each use you make of RapidSMS.

What can be done

The following part is meant to give technical details on what and how RapidSMS could be improved.

Turning RapidSMS into a pluggable Django app

RapidSMS replaces several part of the Django framework:

- it changes its behavior, like when it overrides `./manage.py` by `./rapidsms`;
- it adds coupling like with its wrapper for `render_to_response()` you can't avoid to use if you want to access `base_template`;
- it enforces its own implicit conventions, such as grouping all applications in the APPS directory instead of the project dir root or `site-packages`.

This makes uneasy to quickly test, integrate and deploy this tool, but above all, reduce the productivity and quality usually using Django leads to.

RapidSMS should be written as an independent Django app so it no longer interferes with its environment. Technical benefits would include:

- The possibility to install it using `easy_instal`, making deployment much easier, being friendly to professional tools such as `virtual_env`, `pip` and `Fabric`.
- The ability to make it work with other apps in a unobtrusive way, making it more adaptive to any need and preventing bug hunting occurring when you sometimes try something innovative or unexpected.
- A total separation from the Django code base, so projects can benefit from new Django and RapidSMS versions/patches. These is critical for security fixes, an important issue since we are already often dealing with medical data.
- The opportunity for anybody to create an app to extend RapidSMS itself, opening it to any other actor than would like to make its own service compatible: SMS gateways, queuing and load balancing systems, external messaging systems, etc.

Plus, this would make installing, testing, learning, and eventually adopting RapidSMS much more quick and easy for beginners. We could then count on a larger pool of quality developers to keep improving the tool without investing more, and of course finally hire as much competent staff as the next projects will require.

Make RapidSMS use Django tools

Rapid SMS now runs while ignoring even basic Django standard tools, using its own instead. As a result, you can not capitalize on Django skills to use RapidSMS, nor can you rely on Django advanced features. Substantial improvements could be made:

- Turn the `.ini` configuration file back to a `settings.py` files, with a separate `settings.py` for the RapidSMS app default settings.
- Create an urlconf like file for SMS routing, something like `pattern.py`, that would match a SMS pattern and would call the appropriate view.
- Make `Reporter` inherit from `User`, so it can uses permissions.
- Get rid of wrappers that forbid any granular control on your response.
- Use views and templates for SMS instead calling everything from `app.py`.

- Use signals and middlewares, instead of calling sequentially the `handle()` method of all applications listed in the `.ini` file.
- Make the polling backend the default backend to send SMS so there are always queued, prioritized and can be accessed outside the SMS router thread.

Create a documentation and push conventions/best practices

There is now no official way of doing anything using RapidSMS. Anybody does whatever works for him, sometime coming up with very messy solutions. Teams across the Word can not share the product of their effort because there are not compatible. Without mentors, beginners have a hard time doing the simplest thing since the only to learn to use it is to read the code, and do a hundred of tries and errors.

Documentation doesn't need to be compulsive, but the main API should be stable, official, and clearly documented so anybody can start playing with RapidSMS right away. Common tasks such as:

- sending an SMS,
- reacting to an SMS,
- testing,
- installing,
- deploying,

Should be illustrated with “how to”s so hiring would no longer be such an issue.

The documentation should mention conventions such as naming, file organization and design patterns, so teams working on different countries are eventually on the same page and can share later their experience and work at a low cost.

Best practices, and related tools, should be highlighted for the same reasons. Installation and deployment would strongly benefit from using virtual_env, pip, Fabric, South, Sphynx, Buildout, Paster and so on. Because many other projects in the word use them, there are a source of quality, productivity and reliability.

Provide sane and useful default

RapidSMS is very versatile and therefor suitable for a lot of different tasks, but has, as well, many different ways of letting you do the same thing. Because of the lack of convention and documentation, anybody has its own strategy to solve a problem.

Let's look at the simple action of sending a text message. You can do it in several ways:

- Directly in the `app.py` file, using the router reference and send instantly a SMS;
- Use the `HTTP` backend and send a `POST` or `GET` request to trigger message sending.
- Use the polling backend and stack messages in the data bases which are processed one by one.

The first one only make sens if you have high priority messages to send, but since there is no urlconf like routing system, everybody uses it to respond to incoming SMS. The second one is generally used for triggering event from the Web interface, but since it doesn't return a Response object, it can be used only we AJAX. The third one should be the natural way of doing it because it allows sorting, setting priorities, pausing and is less resource consuming. But it is not well known and little used.

The database queue is the most flexible and universal one, and yet the less used. Reniewing RapidSMS would involve promoting the use of a sane default and obvious way of doing things,

while the other one should remain possible. With the previously suggested architecture, sending messaging should always be done via a message queue, whereas in the few cases where you would need a shortcut, you could use middlewares. HTTP backends would not even make sense with a proper MVC design.

What's more, several parts RapidSMS are often rewritten such as permissions (as Reporter let you no way to do it) and logging (the current one being very hard to relate to any event). RapidSMS should provide default flexible API for managing such com mun tasks.

Improve debugging tools

Improving RapidSMS will be a huge step, but eventually no organization will want to dedicate ad vitam time and budget to maintain it. As an open source tool, it is natural to think that the community will then be in charge of its future. The only way to ensure that is to build in in a way others can understand and trust.

Understanding will come with a proper documentation, but trust require the code base to be heavily tested. Automated tests is the only reliable way for a third party to check if its environment is a got fit for RapidSMS and vice-versa. Unit tests will allow the community to modify the code, improve it, add features and correct bugs and still been able to verify it what they have down follow the commonly expected behavior. Tests means standardization, aggregation and finally, durability.

Among all the biased modification RapidSMS applies to Django, breaking the testing tools in one of the worst. Working with an untested RapidSMS is risky, but experience of others can tell you that it's possible. But working on a new code without tests is having no safety net. When support means hiring somebody 2000 miles away and send him in a tropical area with Malaria and no power, you want to limit it as much as possible, and test your code.

RapidSMS test abilities must be fixed and made easy to use.

Some other issues could nicely improve the day to day debugging practice, such as:

- Fixing out of context errors messages. e.g : An import error can trigger `settings.py not found`, which is not helpful.
- The router should auto reload modified modules in debug mode.
- Improve tools such as the messaging application used to test SMS: removes bug, include the the entire stack trace if there is an error, change layout to make it more obvious to non techies so they can use it to, add a way to save, import and export sessions to replay them like a scenario.

Conclusion

It's especially because RapidSMS promises to be a core technology for information management in third world countries that it should be a priority to ensure it remains a success story.

Hiring, training and maintenance could soon become major bottlenecks in any project involving this technology if no effort is made to make it evolve into a more flexible and standard tool.

Among other benefits, such a transformation would eventually lead to quicker and/or cheaper developments, and ensure that it will be possible for locals to take the lead once the system is set up.