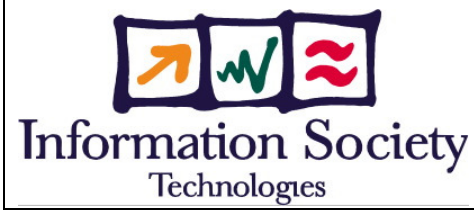
OPAALS PROJECT

Contract n° IST-034824

WP10: Sustainable Community Building

Del10.12 - Component-based visualisation system with collaborative OKS core scenarios



Project funded by the European Community under the "Information Society Technology" Programme

Contract Number: IST-034824

Project Acronym: OPAALS

Deliverable N°: 10.12

Due date: Month 36

Delivery Date: Month 36

Short Description:
In this deliverable we report Wille2 Visualisation System, a component-based visualisation system that can be configured on top of peer-to-peer (P2P) network. Collaborative OKS core scenarios and their implementations on Wille2 are also described.

Authors (in alphabetical order): Jukka Huhtamäki (Researcher), Ossi Nykänen (Senior Researcher, Team Leader), Jaakko Salonen (Researcher)

Partners contributed: IPTI, SUAS, TechIdeas, UniKassel

Made available to: OPAALS Consortium

Versioning

Version	Date	Name, organization
v1	03/09	TUT internal draft
v1.5	04/09	Intermediary comment draft
v2	05/09	Consortium internal comment draft
v3	06/09	Final version

Quality check

Internal Reviewers:

Fernando Colugnati (IPTI)

Dependences:

Achievements*	Implementation and description of a component based visualisation system with collaborative OKS core scenarios
Work Packages	• WP3: visualisation and visualisation system requirements for P2P network models • WP5: technical specification for and proof-of-concept implementation of component-based visualisation system • WP6: Framework for visualisations in evolutionary P2P knowledge space • WP10: P2P framework for OKS visualisation
Partners	IITK, IPTI, UniKassel, LSE, SUAS, UNIS, TechIDEAS, WIT
Domains	• Computer Science: Technical implementation • Natural Science: Technical requirements, vision • Social Science: Process models, collaboration scenarios, social network analysis case
Targets	• P2P infrastructure developers • Partners and third parties implementating visualisations (visualisation applications or components) • Computer Science Community • OKS visualisation users
Publications*	Nykänen, O., Huhtamäki, J., Salonen, J., Pohjolainen, S., & Silius, K. (Eds.) 2008. Proceedings of the 2nd International OPAALS Conference on Digital Ecosystems: OPAALS 2008. 7-8 October 2008, Tampere, Finland. 108 pages. Available at http://matriisi.ee.tut.fi/hypermedia/julkaisut/opaals2008-proceedings.pdf (PDF 3.3 MB) Huhtamäki, J., Nykänen, O., & Salonen, J. (2009b). Catalysing the Development of a Conference Workspace. In HCI International 2009 Conference Proceedings, San Diego, CA, USA, July 19-14, 2009. Berlin/Heidelberg: Springer. To appear. Nykänen, O. 2009. Semantic Web for Evolutionary Peer-to-Peer Knowledge Space. In Birkenbihl, K., Quesada-Ruiz, E., & Priesca-Balbin, P. (Eds.) Monograph: Universal, Ubiquitous and Intelligent Web, UPGRADE, The European Journal for the Informatics Professional, Vol. X, Issue No. 1, February 2009, ISSN 1684-5285, CEPIS & Novática. Available at http://www.upgrade-cepis.org/issues/2009/1/upgrade-vol-X-1.html Nykänen, O., Salonen, J., Haapaniemi, M., & Huhtamäki, J. 2008. A Visualisation System for a Peer-to-Peer Information Space. Proceedings of OPAALS 2008, 7-8 October 2008, Tampere, Finland. pp. 76-86. Available at http://matriisi.ee.tut.fi/hypermedia/julkaisut/20080807-nykanen-etal-p2pvis.pdf (PDF 0.6 MB)

PhD Students*	Jukka Huhtamäki: Researcher
Outstanding features*	Description, tutorial and implementation of a component-based visualisation system, OKS visualisation process models and collaborative core scenario descriptions
Disciplinary domains of authors*	Computer Science: Jukka Huhtamäki, Ossi Nykänen, Jaakko Salonen

The information marked with an asterisk (*) is provided in order to address Recommendation n. 4 from the Year 2 review report

Table of Contents

1. Introduction

In this deliverable we present a component-based visualisation system with collaborative OKS core scenarios. In brief, this deliverable provides a concise introduction to the work topic, report the key results, provides references to the related publications and software artifacts, and outlines and discusses the included demonstrative applications. The work builds fundamentally upon Deliverable 10.6: A proof-of-concept implementation of a visualisation client (Haapaniemi, M., Huhtamäki, J., Kortemaa, A., Mannio, M., Nykänen, O. & Salonen, J. 2007) and Deliverable 10.11: Specification of the P2P configuration of the visualisation system (Huhtamäki, J., Nykänen, O., Salonen, J. 2009a). Due to this inherited relationship, it is assumed that the reader is to some extent familiar with this previous work. In particular, we do not repeat the information visualisation data state reference model (data transformation-visualisation transformation-visual mapping; see Illustration 6 in D10.6) or the survey of visualisation system and adapted components (see Appendix E in D10.6).

The main contribution of this work lies in the introduction of redesigned, version 2 of Wille visualisation system that enables rapid prototyping and efficient implementation of data-oriented visualisations. *Wille visualisation system version 2* (*Wille2* in short) implements the main concepts of a component-based visualisation development framework, providing general-purpose tools for visualisation developers. We introduce the technical core implementation of the system and provide examples written using high-level scripting language, Python. The example applications demonstrate the composition of complex data and visualisation processing applications by invoking a sequence of integrated service components.

In short, Wille2 provides tools for accessing (and publishing) data and visualisation services, processing data in abstract objects, and deploying visualisations either as stand-alone applications, as integrated components within the OKS Desktop application, or as "Webised" services. To support maximal integration, the Wille2 framework is designed to be exploited with respect to three architectural configurations: as a processor, as a hosted component, or as a (HTTP) adapted component. This provides the necessary technical flexibility for developing and integrating visualisations with different technologies and operating systems.

It should be emphasised that, as such, Wille2 is not an end-user product --- it is mainly aimed for developers creating end-user products (visualisation and/or data processing applications). However, to support abstract data processing and visualisation application development with a high-level graphical user interface, the deliverable includes an integration of the framework with the visual programming tool Orange Canvas. In particular, this allows working with pre-processed in a reasonably intuitive and explorative way. Also, to demonstrate Wille's applicability for creating end-user visualisation products, scenarios in Chapter 5 present use cases of Wille that are mostly useful for end users as such.

Finally, the deliverable is "finished" and can be fruitfully exploited as it is. However, the Wille2 visualisation framework is supposed to be used in the context of an appropriate (OKS) Service platform, providing the services of user identity/access management, semantic search, and scalable visualisation service deployment. While the Wille2 framework *is* in itself to certain extent capable of providing all of these functions, best developer/user experience is obviously achieved in a context of all kinds of (OKS) services --- not just the ones specifically set up for visualisations.

Further, it is assumed that the development of the Wille2 framework still continues in the third and the final planned OPAALS project phase. Reasons for this are twofold. First, at the time of publication of this deliverable, the OPAALS OKS Peer-to-peer infrastructure was not yet deployed in a stable and documented way. This means that while the Wille2 can access and publish Webised services, integration with the OKS infrastructure could not yet be finished. Second, as the OPAALS OKS opens publicly, most of the OPAALS stakeholders will not be developers: End-users need

end-user visualisation applications. Thus, in the third OPAALS project phase, the Wille2 framework will be used for developing a series of generally useful OKS visualisation applications. This should also significantly broaden the audience of our work.

The following three appendices complete the deliverable:

- Appendix A: Developer's Documentation and Catalog of Wille2 Orange Widgets
- Appendix B: Complete example listings of OPAALS 2008 data
- Appendix C: Wille2 HTTP Benchmarking

1.1. Resources and Applications in this deliverable

The key software artefact for this deliverable is the second version of Wille Visualisation System. Wille 2 Visualisation System Bundle (In short: Wille 2 bundle) is a software package that includes Wille2's Core implementation (Wille2 Library) and important resources and applications for the context of this deliverable. Wille2 Bundle is available for download in Wille's homepage, at:

```
http://tut.fi/hypermedia/en/publications/software/wille/
```

Firstly, the visualisation applications (or *apps* in short) available in Wille2 bundle are the following:

- `fileserver`: Fileserver (for usage and description, see sections 4.5.4 and 5.5)
- `html-keywords`: Example of a full visualisation application (see description, section 4.5.2)
- `opaals-wiki-sna`: OPAALS Wiki Social network analysis data collector (see scenario in section 5.2)
- `orange`: - Orange Canvas components for exploring data and visualisations by incremental and visual means (see scenario in section 5.3)
- `search-dashboard`: A proof-of-concept dashboard implementation, based on Visual Web Search (for description, see section 5.6)
- `viswebsearch`: An implementation of Visual Web Search (for description, see section 5.4)

Secondly, Wille2 bundle includes a set of services and their required components. A summary of the services and components is available in table 1.1.1.

Table 1.1.1. *Summary of services and required components available in Wille2 bundle*

Service name	Description	Used component(s)	Wrapping method
`geopip-hostip`	Service that translates IP addresses to a geographic location	hostip.info web API	HTTP GET
`img2thumb`	Service for convering a resized version (thumbnail) from a given image locally	A python script + PIL (Python Imaging Library)	Command-line (isolated)
`itemise-local`	Simple, experimental local filesystem itemiser for itemising files to 3D space	A python script	Command-line (isolated)
`opaals-wiki-search`	Service for listing resources from wiki.opaals.org by specified criteria	A python script	Command-line
`pdf2txt`	Service for extracting textual data from a PDF file by using	Multivalent (Java)	Command-line (isolated)
`resourcefetcher`	Service for fetching given list of resources from any MoinMoin wiki with provided credentials	Custom Java component	Command-line
`resourcefetcher-1file`	Single file variant of resourcefetcher	Custom Java component	Command-line
`sna.log.graph`	Service for creating graph representations (Pajek or GraphML) from an XML event log with a component	A python script	Command-line
`sna.log.matrix`	Creates a matrix representation of SNA data	A python script	Command-line

	on basis of an event log. Supports text, HTML and XML.		
`tidy`	HTML validator + cleanup service	HTML Tidy (exe)	Command-line (isolated)
`wiki.historyeventlog`	Creates an event log on basis of wiki editing history.	Saxon 8 (Java)	Command-line
`wiki.pagehistory`	Scrapes the editing history information from an HTML-formatted history page of a wiki. Supports MoinMoin.	Saxon 8 (Java)	Command-line
`xslt-saxon`	XSLT 2.0 service (Saxon-based)	Saxon 8 (Java)	Command-line
`xslt-xalan`	XSLT 1.0 service (Xalan-based)	Apache Xalan (Java)	Command-line

Thirdly, many open sources components were wrapped to Wille 1 infrastructure - a comprehensive list of these components can be found at Appendix G of Del 10.6. In Wille2, some of these components become obsolete; it readily provides easier and more efficient ways of performing similar tasks. The following components, however, could be integrated to Wille2 with some extra work:

- *Data import components:* `get_wiki_attachments`
- *XML processing components:* `aggregate_xml`, `convert_rss`, `validate_xml`, `run_xquery`
- *Semantic Web components:* `aggregate_rdf`, `foaf_creator`, `foaf-o-matic`, `rdfxml2tree`, `sparql_query`, `sparql2vector`, `sparqlxml2map`, `sparqlxml2timeline_data`, `swdesk2`
- *Graph visualisation components:* `graph2svg`, `histogram2graph`, `interestcommunity`
- *Mathematical modelling components:* `matrix2mathml`, `pagerank`, `run_octave`
- *Tools for textual data mining:* `texttools`
- *Web publishing components:* `tpl_engine`, `webdav_publisher`, `webstart_creator`

Finally, several components/wrappers were implemented to Wille 1 that could be - with some extra work – integrated to Wille 2 as viewers:

- `3DForest`: Depicts XML information sources as 3D trees
- `graphl`: Displays RDF as a graph
- `graphshow`: Displays GraphML as a graph
- `somviz`: Self-organising map library
- `squiggle`: Displays SVG
- `vizster`: Community visualisation player
- `zgrviewer`, `zgrviewer_applet`: Visualises SVG or GraphViz graphs

2. Background

Before introducing the Wille2 visualisation system in detail, we shall next briefly discuss the scope, context, design rationale, and challenges of this work. This should help not only positioning the results appropriately, but also understanding some of the pivotal research and design decisions.

2.1. Scope

The underlying objective of information visualisation is to serve as an amplifier of the user's cognition through expressive views giving insight on a certain phenomena represented by the data (see, e.g., (Ware, 2004; Geroimenko & Chen, 2005; Telea, 2008)). Both the visual information seeking mantra ("Overview first, zoom and filter, then details-on-demand[;] Overview first, zoom and filter, then details-on-demand[, ...]") by Shneiderman (1996) and the Card-Mackinlay-Shneiderman (1999) visualisation reference model suggest that visualisation is an interactive process rather than a static image. At best, a visualisation user is able to interact with all the levels of visualisation process from data transformations and visual mappings of data to interaction with the final visualisation view running on a software (cf. Card et al., 1999). Also Kosara, Hauser and Gresh (2003) stress the importance of interaction: "interaction is particularly important in InfoVis: for exploration, analysis, and presentation of allows the user to implicitly form mental models of the correlations and relationships in the data, through recognition of patterns, marking or focusing in on those patterns, forming mental hypotheses and testing them".

Information visualisation is a main operator in different processes including knowledge crystallisation (Card et al., 1999), visual data mining (cf. Keim, 2002; Witten & Frank, 2005) and knowledge discovery in databases. Due to the significance of the application domain, several different approaches to information visualisation exist. Mainstream Web applications such as Yahoo Pipelines (http://pipes.yahoo.com) and IBM Many Eyes (http://many-eyes.com) are, in practice, doing a good job in demonstrating the concept of collaborative, distributed data-processing and data-driven visualisations. Moreover, several general-purpose visualisation tools including Mondrian (http://rosuda.org/mondrian/) and others are available for use as well as visualisation-oriented tools dedicated to different domains ranging from social network analysis (Pajek, Vizster) to visual data mining (Orange). Tailor-made visualisations can be developed with the support of visualisation frameworks including Prefuse (http://prefuse.org), Processing (http://processing.org) and others (http://www.visualcomplexity.com/). Connecting the individual tools to the data sources is, however, yet difficult as well as the co-ordinated use of different tools.

Due to the increasing popularity of Web mashups, the availability of visualisation data has improved in recent years. A large number of Web APIs exist, providing data for mashups and visualisations in explicit formats. Examples include Google Maps API (http://code.google.com/apis/maps/) for geographical data and Audioscrobbler (http://www.audioscrobbler.net) for data representing the listening habits of the users of an online radio service Last.fm. Moreover, Extensible Markup Language (XML) and other text-based data formats enable the utilisation of personal data in visualisations in a more straightforward manner than the proprietary binary formats. The latest text document and spreadsheet formats including Office Open XML and others, for example, are based on XML. At best, a visualisation user could fuse her private data into generally available data in a data secure manner.

Interestingly, major companies have also partly opened their infrastructure for developers. For instance, Google App Engine (http://code.google.com/appengine/) and Amazon Web Services (http://aws.amazon.com) can be used to build RESTful applications for processing and visualising data. However, from the perspective of a Peer-to-peer knowledge space, relying upon a central service might not be accepted. Further, the requirement of establishing a large-scale service for

creating grass-roots visualisations narrows down the application areas. Local and contextual visualisations are built upon local data, perhaps from the developers' computers, and local decentralised community archives (acting as P2P nodes).

2.2. Context

As a visualisation system -- a technical framework for implementing visualisation applications -- Wille2 implements the following central (system) use cases/requirements for visualisation developers:

1. **Pipeline data processing.** It is possible to read data from multiple sources, process data in several internal processing steps, and output data in a suitable format for external activities/interactions.
2. **Service composition.** It is possible to realise individual data processing steps as requests to (external) services.
3. **Webised applications.** The system is used in network applications. In particular, this implies that the system is able to access network data and services, but also that it is possible to configure and distribute visualisation applications in various ways (as services, stand-alone applications, or components in third-party applications).

More abstract (business) use cases for end-users were compiled from the OPAALS consortium in the beginning of the project (see Nykänen, Mannio, Huhtamäki, Salonen, 2007). These included: 1) Navigation aid; 2) Project structure visualisation; 3) OKS structure visualisation; 4) Research topic visualisation; 5) Project partner network visualisation with geographical data; 6) Temporal activity visualisation; 7) Linkage from OPAALS to the external world; 8) Data manipulation interface; 9) Link annotation tool; 10) Semantics of Business Vocabulary and Business Rules (SBVR) vocabulary visualisation. Note that these include visualisation applications. In this setting, the visualisation system is perceived as enabling technology.

Wille2 visualisation system aims balancing the needs of the current and the future network infrastructure and applications. This means essentially three things:

- Wille2 is designed to nicely integrate with the current client-server Web applications. For creating general purpose applications and for adoption, this is a strict requirement. Also, motivation for, e.g., designing new kinds of visualisation applications --- or developing new kinds of P2P services --- may grow from prototyping or showcase work with the already available services.
- By specification and design, Wille2 is designed to work with P2P applications, providing an extensible implementation for invoking services (Del 10.11). When new kinds of service access mechanisms become available, they can be included within the current implementation. To support grass-roots P2P, Wille2 also provides a lightweight infrastructure for publishing services. This enables, e.g., ad hoc networked visualisation (and other!) applications without dependencies to additional software components.
- Wille2 also enables architectural configurations that support integration with different kind of systems and end-user applications. The most notable example is the OKS Desktop Application (see (Desodt, 2007)). At the time of writing, two particular pieces of software from the OPAALS project fall into this category: Sironta (http://www.sironta.com/) and Guigoh (http://www.opaals.org.br/). Special attention to integration also helps integrating Wille2 visualisation applications with other systems.

From the perspective of the Wille2 visualisation system, OKS is the amalgam of different data sources, services, and processes available. By design, the Wille2 visualisation system follows the specification of the OKS Data Model, which aims formally capturing the data repository aspect of

the OKS (Nykänen et al., 2008b). This possibility for semantically well-established interplay between different applications is important in composing visualisations. If data semantics are not written out, creating visualisations become tedious and automating data-driven visualisations becomes very difficult. The scenarios developed for this deliverable clearly demonstrate that good data quality and ability for mechanical pre-processing are pivotal in designing visualisations.

2.3. Design Rationale

When the Phase II of the project started, we outlined the initial conceptual architecture and design rationale for the Wille2 visualisation system, based on pre-studies, discussions, and prototyping (for details, see (Nykänen, Salonen, Haapaniemi, Huhtamäki, 2008a)). In short, the design rationale highlighted the following observations:

1. Visualisation components should be useful as such, without always having to install a specific pipeline processor client to use them. For instance, accessing a simple visualisation component should be possible by writing a shell script or an HTML form.
2. There should be a way to easily create new visualisation components from the existing data processing applications, and offer these as visualisation service components using an appropriate service platform.
3. Besides P2P, the API of visualisation components should follow the conventions of REST when applicable. For instance, it would be nice to integrate existing mashup and feeds to visualisations, without always having to implement complex adapter or wrapper systems.
4. Visualisation components are not (in general) responsible for storing (intermediate/final) results or their computations – this is the responsibility of the visualisation client. For instance, when passing output of a component A to another component B, A does not (in general) have to maintain temporary resources for B, during computations.
5. Visualisation clients are defined by their ability to meaningfully exploit the services of visualisation components, not by their internal making. When writing new visualisation clients, however, accesses to an appropriate, general-purpose visualisation client framework (e.g. programming libraries for common tasks) is typically helpful.
6. Visualisation clients (acting as pipeline processors) typically fall into three main categories: (command-line) scripting (e.g. Python script/program), local application (e.g. with a graphical user interface), and web application (e.g. AJAX or HTML user interface)

As suspected, besides actual application development, we have developed, exposed, and tested our ideas in a series of publications of the related areas (Nykänen, Mannio, Huhtamäki, Salonen, 2007; Nykänen, Salonen, Haapaniemi, Huhtamäki, 2008a; Nykänen, Huhtamäki, Salonen, Pohjolainen, Silius, 2008). A significant subtopic, relationship with formal knowledge in the OKS, has been extensively discussed in (Nykänen, 2007a; Nykänen, 2007b; Nykänen, Salonen, Huhtamäki, Haapaniemi, 2008b; Lapteva, Peukert, Nykänen, Eder, 2009; Nykänen, 2009). In particular, we have found it useful to perceive the OKS as a contributed knowledge system (see (Nykänen, Salonen, Huhtamäki, Haapaniemi, 2008b)), with characteristics of Data-Sharing Systems (Smith, Seligman, & Swarup, 2008). Emerging collaborations include aligning our work with the visualisation service for P2P infrastructure and EvESim by SUAS (see (Kurz, Heistracher and Eder, 2008)).

While the design rationale explains many of the design decisions, it in itself needs some explanations. The bottom line is that components of visualisation applications are developed in iteration. Further, data and viewer (or browser) components provide significant dependencies and requirements for the work. Visualisation applications also benefit from abstraction and

encapsulation. A good (service) component can be utilised in several different ways, perhaps in different contexts.

The design rationale also hints the role of the P2P networks in visualisations (about P2P, see, e.g., (Steinmetz, R. & Wehrle, 2006)). The OKS can be used for recording (and searching) metadata descriptions of the available data, services, and applications, P2P and other. Thus, whether a visualisation itself is P2P or not, is determined not only by its deployment, but also by the services it requires. This realistic design stance allows using the same technology for implementing P2P and other visualisation applications. It also means that not all visualisations implemented using the Wille2 visualisation system are truly P2P. For instance, relying onto a centralised service, such as Flicker, may obviously introduce a risk of central point of failure. Also, the P2P property may be lost because of poor design, e.g., using a specific service when a similar distributed service is available. However, it is important to understand that centralisation takes place in most business processes: Some data or tasks simply cannot be duplicated either for physical (e.g. manual work, use of specific devices) or business reasons (e.g. protected activities or products).

2.4. Challenges

Obviously, developing a general-purpose visualisation system is not easy. In order to understand the scope and the output of this deliverable, few challenges are worth considering briefly.

1. **Visualisation system vs. visualisation application.** In brief, a visualisation system sets the basic concepts and tools for creating visualisation applications efficiently. On the end-user (or technically superficial) level, it is easy to confuse these two related concepts. However, typical visualisation applications share several design components and architectural principles. This challenge, sometimes as a mismatch of expectations, gets typically highlighted in communication with non-technical audience since the best way to talk about the properties of the visualisation system takes place via specific visualisation application requirements and examples.
2. **Visualisation vs. user interface.** Most applications include components for showing data to the user. On the other hand, a strict definition of scientific or information visualisation might not include all the components we consider as visualisations (for instance, a data-driven table of contents of conference proceedings). This means that the borderline between visualisation applications and applications with some user interface component is a bit blurred. We feel that strict distinction between visualisations and other applications is not necessary: Visualisations are characterised by their design (use of visualisation tools, pipeline design pattern), sensitivity to changes in the source data (data behind visualisations has an explicit nature), and the idea of capturing selected aspects of the data with respect to some audio/visual/haptic (etc.) rendering model (all source data may not be presented in its original form).
3. **OKS definition vs. deployed OKS infrastructure.** In brief, OKS exists in various levels on abstraction in the project. However, an abstract definition does not as such yield "off-the-shelf" infrastructure. Indeed, one of the main challenges of developing the Wille2 visualisation system deliverable has been the fact that the detailed development of OKS as (P2P) infrastructure technology has taken place parallel (and at the time of writing is not finished). This means that while the abstract concepts of the underlying OKS infrastructure are relatively well understood, the visualisation system deliverable cannot exploit intuitive features of the conceptual OKS until they are implemented. However, visualisation applications, services, etc. are simply certain kinds of OKS applications, services, etc., requiring the common infrastructure. Of course, this breakdown of tasks is not strict. In particular, we have made a significant extension to this setting by including a lightweight

service deployment feature to the very core of the visualisation system, in order to increase its independent value.

4. **Relationship with other visualisation systems.** As briefly pointed out in the beginning of this section, even if not directly overlapping with the Wille2 visualisation system, several other independently developed visualisations systems exist. Rather than aiming to compete with the existing systems, the Wille2 visualisation system aims integrating with them. This sets another moving target for the work. However, by default, the component-based design and basic Webising ideology seems to support this interplay well. Nevertheless, aiming for a service-oriented general purpose system means that more specific applications may no doubt outperform Wille2 in their home fields. (Btw. This is an instance of a generalisation of the well-known Contingency theory, or hypothesis: While different approaches can be compared in a specific application, there is no single best approach for all applications.) For instance, while it is possible to implement "pull" kind of applications using the Wille2 visualisation system, application with unknown response times, by default works best with a "push" approach. This is a general property of a complex pipeline kind of systems: Computing complex pipelines composed from abstract (and thus e.g. quite slow) services becomes quickly very expensive.

While some of the above challenges are severe, we believe that with the collaborative work of the consortium, we have been able to overcome these challenges well. As the deployed OKS infrastructure matures, we shall refine the technical OKS integration accordingly. This work is captured as an additional work item in Phase III.

3. General Description

Having established the context of our work, we shall next introduce the main user role and system concepts. In turn, these help in understanding both the technical design, the related processes, and applications of the Wille2 visualisation system.

3.1. User Role Concepts

The deliverable Del 10.6: A proof-of-concept implementation of a visualisation client introduced the basic roles of visualisation system users: Visitor, (End) User, Designer, Developer, Administrator, and Content Author. In this deliverable Del 10.12 we have narrowed the treatment down to four essential roles: (Content) Author, (Visualisation Component) Developer, (Visualisation Application) Designer, and (Visualisation Application) End-User:

- **Authors** produce, adapt, or deploy accessible content. For instance, an author might write a wiki page, add calendar event to a public collaboration space, or publish a document in a Peer-to-peer file service.
- **Developers** implement, wrap, or deploy services suitable for composing visualisations. For instance, a developer might provide a new service or component (e.g. a service outputting calendar data from a content management system) available to others, or simply implement a suitable software wrapper to an already existing software component (e.g. a linguistic analysis tool).
- **Designers** implement, compose, or configure visualisation applications. For instance, a designer might configure a visualisation application service to suit a particular application purpose, or, exploiting some existing services, write a Python script outputting a graphical End-User interface.
- **End-Users** utilise visualisation applications in their activities. For instance, a researcher might use a social graph view in order to review the work and contact other researchers of a particular domain.

In practice, the roles of Developer and Designer may significantly overlap. In order to simplify our terminology about roles, we mainly speak about developers, understanding that developers are in general capable of acting as designers, but not necessarily vice versa (due to the required technical skills). Also, several roles may apply to organisations or individuals simultaneously. In particular, End-Users might also appear as Authors, and vice versa.

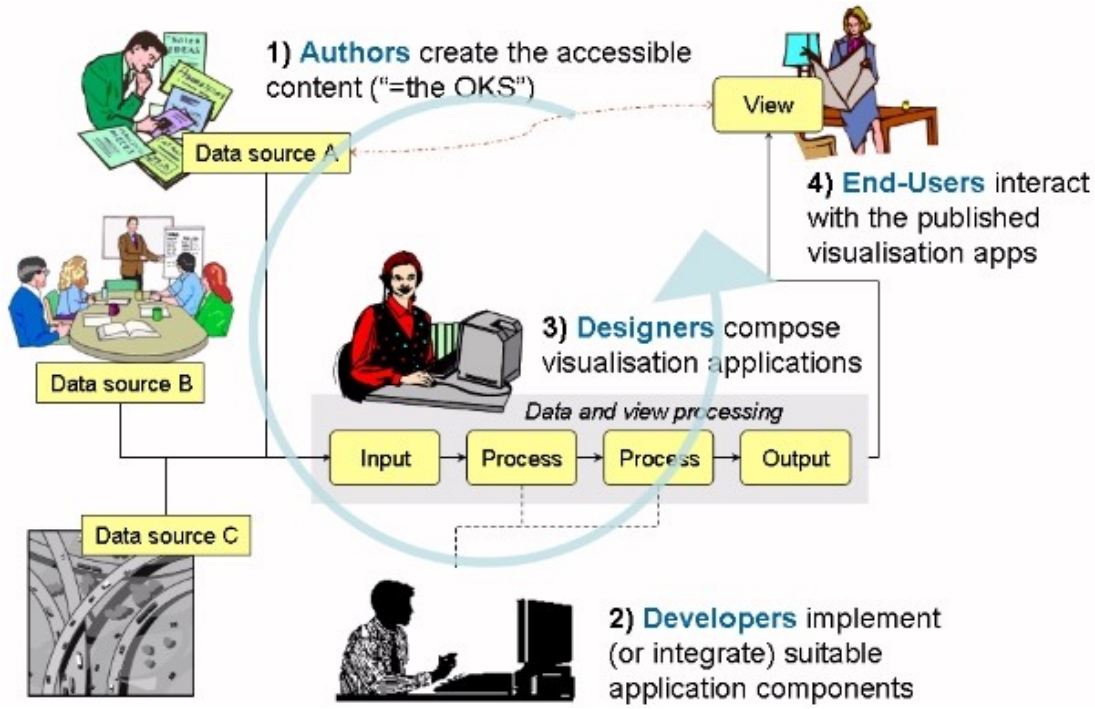


Figure 3.1.1. *Basic iterative collaboration cycle of creating visualisations*

Figure 3.1.1 (above) depicts the basic iterative cycle of creating visualisations. The basic idea is that the process provides a natural feedback loop between different user roles. In particular, visualisations may include references to authoring practices. This allows integrating view activities with authoring, even if visualisation application per se might not include authoring tools.

Note also that the process includes the common Chicken or the egg causality dilemma. This means that in order to be successful in practice, a certain critical mass of content, components, and visualisation applications ought to be available. For instance, motivation of authoring/publishing certain kind of content might realise only when the content plays a role in useful visualisation applications.

3.2. System Concepts

For purposes of this document, it is useful also to characterise the basic terminology of concepts and system components related to visualisations. The basic idea is that **Visualisations** are conceptual systems understood by humans. Visualisations are implemented or realised using **Visualisation Applications**, essentially as software components executed by computers. Rather than individually, visualisation applications can be cost-efficiently implemented using the basic concepts and tools provided an appropriate **Visualisation System**. (As suspected, the Wille2 visualisation system provides such of a system.)

A Visualisation Application performs some application logic that, based on some input data, typically provides a configurable **End-User Interface** (graphical user interface [GUI] or command-line interface for humans), or useful **Data Ouput** (for computers), representing the data with respect to some useful **Visual Model or Format** (of course, other modalities such as Aural or Tactile can be used as well). In the former case the application provides the End-User interface in itself. In the latter case, the application may "only" output, e.g. a file to be opened in some interactive **Viewer Application**.

Visualisation applications include internal structure. In general, it is useful to identify at least the following components: **Program Logic** and **Service Platform Client**. While the program logic is different in different visualisation applications, service platform clients perform mostly similar kinds of duties: for example request application data and interact with the **Service Infrastructure** (request or deploy data/services). Besides means to publish and access OKS-specific services, such as OPAALS distributed identity model, the Service Infrastructure essentially provides access to the common Web applications as well (as uniform resources, particular Web API services, or common Web Services). As such, the Visualisation System does not address the topics of user management, (Platform) Service security, or P2P identity of resources (which are addressed in other workpackages of the project). These concepts are realised on the implementation level of the infrastructure and individual applications. Further, as the feature set of the service infrastructure matures, access to the provided functionality may be integrated to visualisation system, to ease the implementation of visualisation applications.

While visualisation applications may appear as self-contained software artefacts, some application tasks may be executed using other reusable software artefacts in terms of services. A **Service** is an adapted **Service (or Software) Component** that is accessible via the Service Infrastructure, and implements an appropriate application interface. For instance, several different visualisation applications might request the service computing a similarity measure between two documents from the Service Infrastructure.

An important conceptual design pattern for the Program Logic is the **Pipeline Processor Pattern**. In particular, many visualisation applications may be considered as a series of abstract data and view transformations. When useful Services are available, the Pipeline Processor Pattern becomes very useful conceptualisation in designing and composing Visualisation Applications from Service Components. However, typical Applications include scripting-like program logic, more complex than sole Pipelines. Note that a typical technical pipeline definition restricts the use of scripting and forbids e.g. cyclic pipelines.

Sometimes visualisation applications do not appear to End-Users as singular, self-contained software artefacts, but as integral parts of other applications. In these cases, Visualisation Applications may exist in relationship with **Embedding or Hosting Applications** (including Orange Canvas, Eclipse and OKS Desktop Applications).

As in the case of many applications, intuitively similar visualisations can typically be realised with several different technical approaches/implementations. For instance, a visualisation showing the activities in a working group might be implemented as a monolithic visualisation application simply accessing data and providing a GUI in the level of the operating system, as a visualisation application accessing data and requesting several services and providing an HTML/AJAX user interface for the Web browser, or as a visualisation service outputting a file to be opened in an external Spreadsheet application.

4. Implementation Perspective

Experiments with the Phase I prototype demonstrated that developing visualisations strictly using a pipeline-based technical architecture was suitable only in a certain kinds of applications. In particular, the data and visualisation processing behind several visualisations could, in fact, be most conveniently expressed in terms of typical programming structures (ad hoc data structures, loops, visualisation-specific computations).

Further, while the requirement of developing and deploying service components following a rather monolithic software development process might be required in developing scalable large-scale applications, it was too challenging for grass-roots developers. In particular, recognising the social aspects of visualisation development suggests that community-driven visualisations are first created ad hoc, perhaps based on local needs, and only refined into genuine scalable services of the underlying P2P OKS after critical mass behind the effort.

As a consequence, the core of the Phase II Visualisation System was developed based the concept of scripting. Due to its popularity, properties, and portability, Python was selected as the main scripting language. Also, in order to test the applicability of HTTP to Wille service transactions, and verify that our implementation of Wille server and HTTP service publication, several benchmarking tests were performed with Wille2. For details on the testing, see Appendix C.

4.1. Architecture and OKS Integration

The basic idea is that the Wille2 visualisation system provides an application framework as a Python library, called Wille2 Library, taking care of the basic data transformation, service invocation, lightweight service deployment, and service integration responsibilities.

Since Wille2 aims providing a framework, different kinds of visualisation applications can be built on top it. To illustrate this, several useful example applications are also included in the deliverable (see the following "Scenarios" section.)

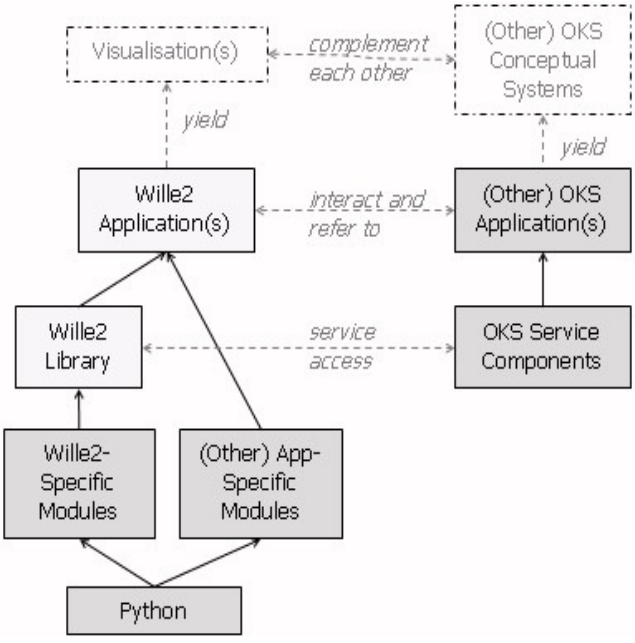


Figure 4.1.1. *Abstract visualisation system layer architecture*

The abstract visualisation system architecture is depicted in Figure 4.1.1. In short, the Wille2 library enables implementing Wille2 visualisation applications that yield conceptual visualisations, utilised in the context of other OKS conceptual systems.

Technical interaction with the OKS takes place on two levels. On the service level, Wille2 applications may access and deploy lightweight OKS services using the Wille2 service and service invocations APIs. On the application level, applications may interact with and refer to each other, perhaps activating each other when required. Note that Wille2 applications may or may not be executed using common OKS technologies. However, since Wille2 applications are implemented in Python, they can be executed either using platform-depended Python interpreters, or with the help of Jython, on top a Java virtual machine. This diversity supports a wide range of deployment ranging from USB memory sticks to centralised Web servers.

Of course, distinction between Wille2 and other OKS applications is not strict: hybrid applications may integrate not only lightweight Wille2 services but also Wille2 modules in their technical design.

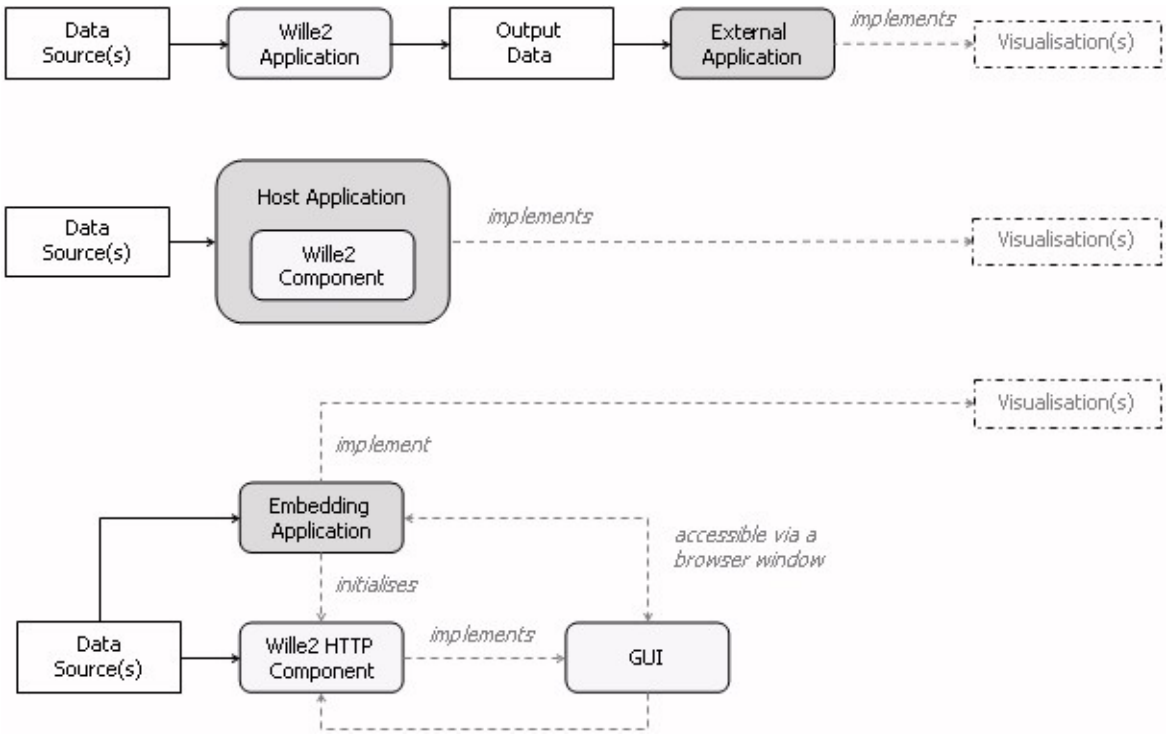


Figure 4.1.2. *Three architectural configurations of Wille2*

To meet the needs of different kinds of visualisation applications, Wille2 may appear in three forms (see Figure 4.1.2). The basic architectural configurations are:

1. Wille2 application acting as a **Processor**,
2. Wille2 application acting as a **Hosted Component**, or
3. Wille2 application acting as a **(HTTP) Adapted Component**.

The first configuration includes a command-line Wille2 application participating as a task in a data processing and visualisation pipeline that outputs data to be opened in an external viewer. Note that that launching the external application (e.g. an interactive viewer) is strictly speaking not in the responsibility of the Wille2 application. An informative use case is creating a makefile or an antfile, in which Wille2 application is executed as a task.

The second configuration includes a graphical Python application that benefits from the functionality of the Wille2 library. An informative use case is creating a library of GUI widgets utilising Wille2 functionality, for some existing application (e.g. Orange Canvas).

The third configuration demonstrates a general strategy of integrating Wille2 visualisation applications in other applications, perhaps implementing with several different technologies and application components. An informative use case is an application that is capable of opening browser windows (or "frames") to show and interact with a (local) Wille2 visualisation application that serves e.g. an HTTP HTML/AJAX GUI. (Note that in principle, the configuration 3 is simple a special case of the configuration 2. However, it is addressed explicitly due to it's practical significance in integration with the OKS Desktop applications.)

It is worth emphasising that, as such, the Wille2 library does not provide a general visualisation editor interface. The abstract method of creating visualisation applications is via scripting. However, in practical visualisation development, graphical developer interface usually aids the process, providing, e.g., file management, graphical pipeline editor, and intuitive widgets for representing tasks. To address the need of developing explorative, data-driven visualisations incrementally, integration with a hosting visual programming editor is included in the deliverable (see Section 5.3).

From the technical point of view, integration with the OKS is based on the functionality of the Wille2 library. In particular, the library provides common OKS service invocation mechanism that is needed for accessing OKS data and its services (including semantic search). Note that the third architectural configuration, (HTTP) Adapted Component, provides a simple way to integrate Wille2 applications with different kinds of OKS applications. In particular, various visualisations can be integrated with the OKS Desktop applications (e.g. Sironta, Guigoh) by integrating things on the level of program logic via an appropriate API, and by integrating things on the GUI level by sharing a browser window or a frame.

4.2. Core Process Descriptions

In the context of the Open Knowledge Space (OKS), it is useful to characterise the main processes that Wille2 applications support. The following process diagrams illustrate the intended development, OKS integration, and usage of visualisation applications.

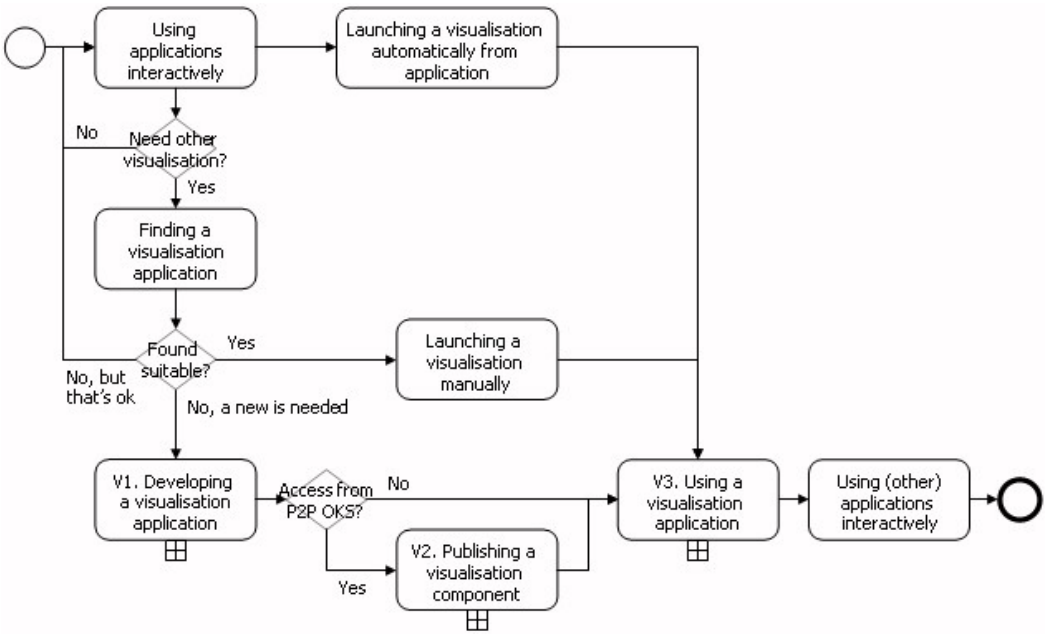


Figure 4.2.1. *Process model V: Visualisation application development and usage*

The process model V above (for Visualisation), outlines the relationship and order of the basic tasks: Using applications, launching existing visualisations, and discovering the need for looking for or designing new kinds of visualisations.

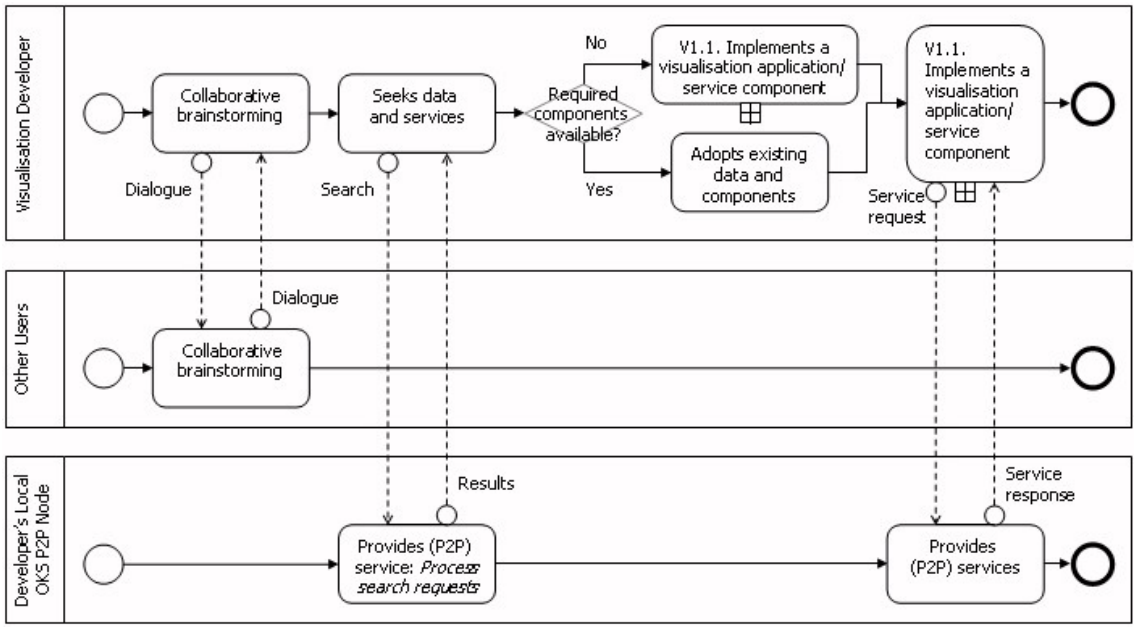


Figure 4.2.2. *Process model V1: Developing a visualisation application*

If suitable visualisations are not available, new visualisation applications can be developed (see figure above). This involves pinpointing the requirements and basic building blocks of the new visualisation, evaluating the applicability of existing service components (if available), and implementing a new (so far missing) visualisation application or service component. Note that the borderline between visualisation services and visualisation applications is vague and is essentially related with deployment: In many cases, intuitively the "same visualisation" can be deployed either as a complex service or application (or both).

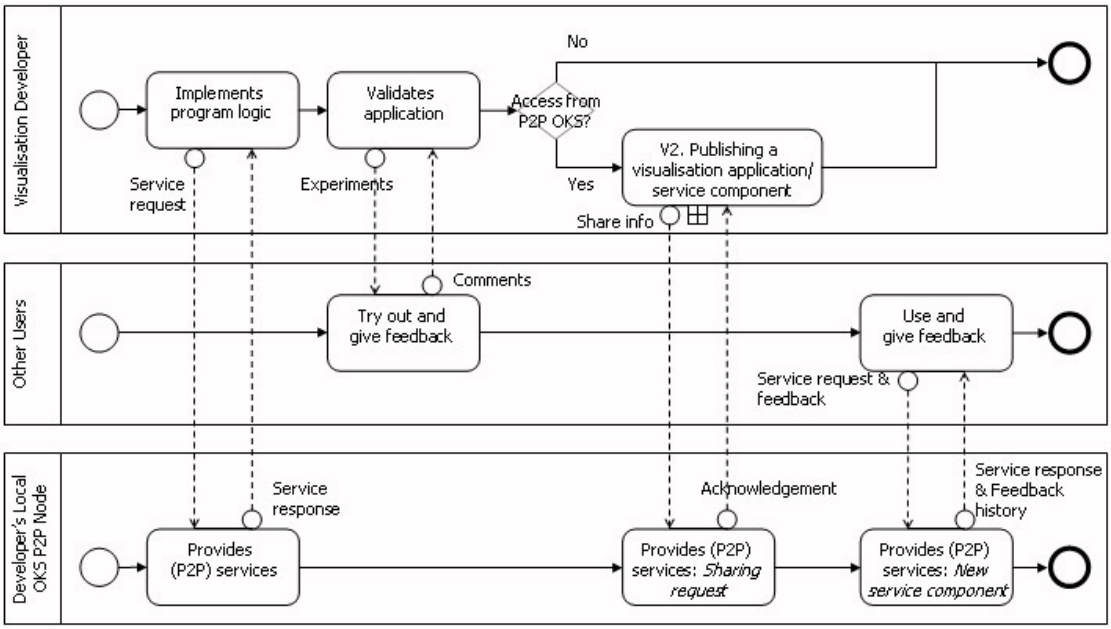


Figure 4.2.3. *Process model V1.1: Visualisation developer implements a visualisation application/service component*

Implementation process (Figure 4.2.3 above) deals with designing appropriate program logic and, e.g., validating a prototype implementation. If the developer wishes to deploy the implementation in the (P2P) OKS, the implementation needs to be, e.g., uploaded or registered with the help of the underlying OKS infrastructure (to enable, e.g., OKS search). Note that applications can be also shared by other means.

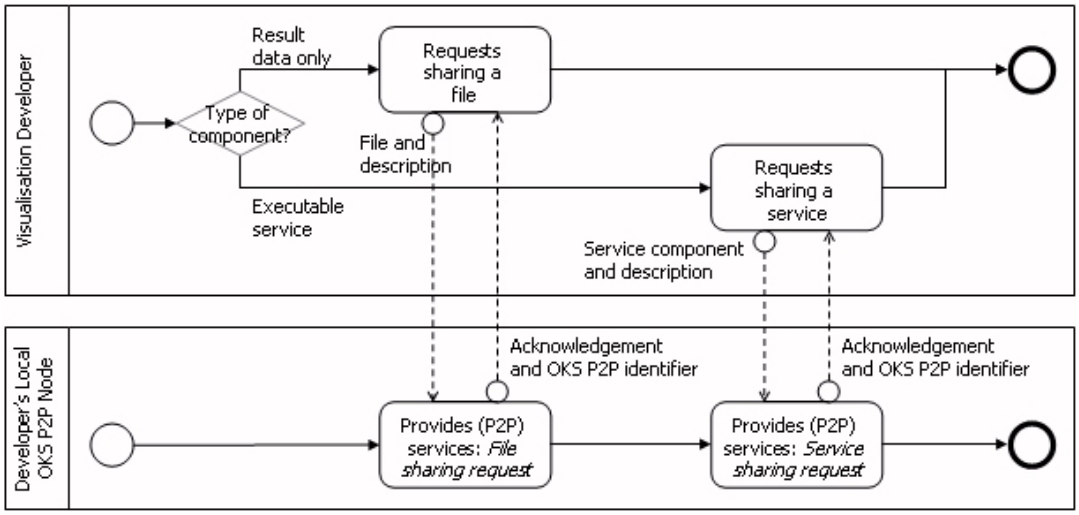


Figure 4.2.4. *Process model V2: Publishing a visualisation application/service component*

Publication involves deploying data and/or service components. It is worth emphasising that while, in principle, both may be captured using the same abstract sharing request, sharing services is technically much more challenging and may require, e.g. using specific implementation/wrapper technology.

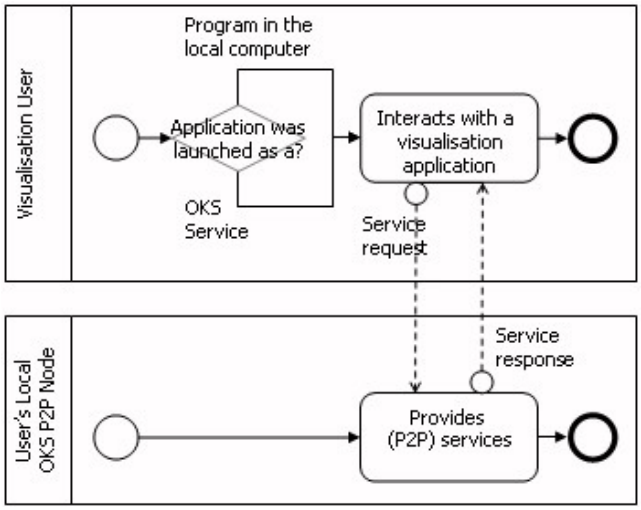


Figure 4.2.5. *Process model V3: Using a visualisation application*

Finally, using visualisation application is straightforward. However, if the visualisation application requires services provided by the common OKS (or the Web), appropriate network access is required.

While the process descriptions are useful in understanding the role of Wille2 applications with respect to services and other applications, these become concrete only after considering examples. Let us next consider some basics of the Wille2 framework in programming detail, and then look at the bigger picture in terms of several usage scenarios.

4.3. Simple visualisation client tutorial

Let us next demonstrate how a simple visualisation client can be constructed with Wille.

Throughout this section, we assume that Wille2 has been correctly set up. For detailed instructions on how to install Wille2 appropriately, please consult the software's `README.txt` file.

4.3.1. Using Wille2 Client

Let us first demonstrate how visualisation developer can use *Wille2 Client* to access existing visualisation services. Wille2 Client is a Python library that can be used to search and execute *Wille2 Services*. Execution of several Wille2 Services can be combined in various ways, in order to create data processing pipelines that can be used to build up visualisations.

Let us first consider a simple example where we use Wille2 Client to execute a single service:

Listing 4.3.1.1 *Example of single step data processing with Wille2 Client*

```
1 import wille.client
2
3 client = wille.client.Client()
4 client.load_dir('../../services/') # Wille2 services path - change if needed
5 xhtml = client.execute_service(name='tidy', \
                          params={'html_data': '<h1>Hello World!</h1>'})
6 if xhtml.success():
7     print xhtml.data
8 else:
9     print "Tidy failed: %s" % xhtml.error
```

If successfully executed, the example code will run HTML Tidy and print out resulting, well-formed markup. HTML Tidy is a software that has been wrapped as a *Wille2 Service* to validate and correct mistakes in given HTML file. The code highlights following tasks that Wille2 Client is responsible for:

- **Lines 3-4**: Local Wille2 Services are loaded from a given subdirectory and made available to the client. Also, services can be loaded from arbitrary number of sources: locally with consecutive *load_dir* invocations or over HTTP with *load_url* calls. There is also room for extension to incorporate other methods, such as P2P network connections, as well.
- **Lines 5-6**: Wille2 client is used to execute a service that matches the given description. Parameters are passed to matching service for data processing. Finally, a result is retrieved and stored as a local variable (tidy). Service execution automatically manages failures caused by typical temporary problems such as casual network timeouts and file locks etc.
- **Lines 7-10**: Result processing and simple error handling. Output data from service execution is available in *data*, whereas metadata about execution can be accessed via *metadata*. *success*-method can be used to check if service has been properly executed.

Wille2 Client also includes the possibility to run a group of services with single call, manage user credentials and launch arbitrary viewer programs. For detailed information on usage, please consult Wille's API documentation.

4.3.2. Creating a Visualisation

Visualisations can be created by utilising data processing pipelines created by combining Wille2 Service calls. Let us next consider a more complete application, where Wille2 is used as a data transforming processor to perform linguistic analysis on a given HTML file:

Listing 4.3.2.1 *Example of a full visualisation application ("HTML linguistics")*

```
 1 import wille.client
 2
 3 client = wille.client.Client()
 4 client.load_dir('../../wille/services/')
 5 opts = '-asxml --add-xml-decl true '+ \
         '--numeric-entities true --output-encoding latin1'
 6 xhtml = client.execute_service(name='tidy', \
             params={'html_data': open('data.html','r').read(), \
                     'tidy_extra_opts': opts})
 7 words = client.execute_service(type='xslt', \
             params={'xml': xhtml.data, 'xsl': open('extract.xsl','r').read()})
 8 topics = client.execute_service(uri='http://www.topicalizer.com/getKeywords/',
 9                                 params={'data.txt': words.data});
10 print topics.data
```

The given listing will read data from local file *data.html* and, in steps, extract important keywords from the HTML content. The steps highlight different features of how to use Wille2 services as follows:

- **Lines 5-8:** Invokes service *HTML Tidy* service by its *name*. Options are prepared to create well-formed XML out of the given HTML.
- **Line 9-10:** XSL Transformer (XSLT) is invoked to select and extract only important content from the data. Note that XSLT service is selected by its type; any matching XSL transformer could be used.
- **Line 11-12:** A keyword analysis is performed on the extracted content. Instead of using a Wille2 Service, *a web API* is invoked (service invocation *by URI*).

In the example, Wille2 Services are invoked successively, all at once. A visualisation may, however, also choose to perform service invocations as required such as only update data when it has changed. Visualisations can also be instructed to execute services in groups, optimally in parallel, if such steps are not dependant on each other.

The example visualisation also uses a third party web API (topicalizer) as part of data processing. This highlights the fact that an important idea in Wille2 is that it can be used to integrate not only existing components but also existing services. This ability should facilitate easier introduction of Wille2 infrastructure, since not everything has to be adapted to Wille. Also, it may be desirable for service providers to retain services and their components under their control: reasons for such could include business, security or legal considerations.

4.3.3. Adapting Components to Wille2

The key to leveraging Wille2 Visualisation System is adapting external components to Wille2 infrastructure. *Components*, e.g. external software that can be used to employ data processing, can be adapted to Wille2 by creating *wrappers* that make them available as *Wille2 Services*.

Let us next consider how we could integrate a component similar to topicalizer as a Wille2 Service. Let us assume that a Java Library, analyzer.jar, is available for us and is able to perform the similar functions by invoking:

```
> java -jar analyzer.jar data.txt -o keywords.txt
```

When ran, *analyzer.jar* reads through data.txt and stores extracted keywords to file *keywords.txt*. The approach works well when the tool is ran manually from command-line. However, in order to

provide the component as a service, it is desirable that no manual work is required to run the analysis.

Wille2 service descriptions are written as wrappers that automatically perform similar tasks. Let us assume that we have created a new directory, *extract_keywords* under *services* folder with Java libraries required for the analyzer. A *willeservice.properties* file that would perform similar commands could be written as follows:

Listing 4.3.3.1 *Example of a Wille2 Component Wrapper (Keyword extractor)*

```
 1 parameters=data.txt, (ignoreList)
 2 adapter=CommandLine
 3 adapter.command=java -jar analyzer.jar @{data.txt} --ignore-list=${ignoreList} -o
keywords.txt
 4 adapter.output=keywords.txt
 5 adapter.output.type=file
```

The properties together construct a wrapper:

- **parameters:** `data.txt` is a required input parameter. An optional list of ignored words (ignoreList) may also be specified.
- **adapter:** The component is adapted by using a *CommandLine* adapter - a quick but unreliable way to adapt a component
    - Given *command* is executed with chosen (CommandLine) adapter with parameters replacing selected parts of the it
    - Component *output* is captured as the content of *keywords.txt* file

Similar to the 3rd party API example, Topicalizer, this service can now be invoked as part of data processing tasks as follows:

```
> topics = client.execute_service(name='extract_keywords', params={'data.txt':
                                                                   words.data});
```

When services are shared over HTTP or a P2P network, useful extra information such as service type and description can be added as required. Use of more specialised adapters, such as IsolatedCommandLine, may also be desirable, since plain CommandLine adapter does not handle parallel executions of a service.

For a more thorough documentation on component wrapper, see Wille's *README.txt* file.

4.3.4. Using an App

Apps are complete applications built on top of Wille2 infrastructure. While any kind of visualisation applications can be built using Wille2 server e.g. over HTTP interfaces, special attention has been paid for supporting App development in Python. As such, there are two possible ways of building apps:

1. by writing an App in Python and serving it on top of Wille2 server or
2. making a standalone App that either requires a running Wille2 server or automatically starts one as required.

Let us use an example App to demonstrate how an App can be used in these two different ways. *Fileserver* is a simple application build on top of Wille2 that can be used to share files over HTTP.

All files placed under *shared* subdirectory will be served. Written in Python, fileserver can be used both as a service on top of Wille2 Server or as a standalone software.

In order to run fileserver as a standalone application:

1. Open shell and change working directory to fileserver's folder (e.g. wille/apps/fileserver)
2. Execute

```
> python fileserver.py 80
```

In order to start fileserver as a part of the Wille2 server:

1. Make sure fileserver has been deployed under Wille2 server's apps folder.
2. Change working directory to Wille's root (such as `c:\Wille2\`)
3. Start server by invoking

```
> python wille-server.py
```

In both cases, port 80 will be used for running Wille2 server. Use *Ctrl+C* to stop the server. Let us assume we have shared `draft.png` by putting it under the shared folder. In order to download it, simply open your browser and enter `http://localhost/apps/fileserver/draft.png` (or corresponding IP instead of localhost) as the URL

By default, fileserver is accessible from localhost only. A second command-line argument can be used to extend server's visibility. A list of IP addresses may be entered to share files only to specific targets. By specifying `public`, the files can be shared to the whole web.

It should be noted that different apps may require different procedures for running. You may be required to read associated command-line documentation or bundled *README.txt* in order to understand how the app can be used, in case it does not provide support for running it on top of Wille2 server.

4.3.5. Creating an App

In this section, we will demonstrate how Wille2 can be used to build an app with a Web-based user ínterface in HTML. As an example, we will create a web-based version of "HTML linguistic" visualisation, presented in section 4.3.2.

Let us create a folder for the app, called html-keywords. By using the "HTML linguistic" application as a base, we write our App in Python. In this case, the following steps need to be followed:

1. Create and implement a handler class for processing incoming GET/POST requests
    1. GET will display an HTML form with input field for HTML file upload.
    2. POST will be used to process file submission and perform actual HTML linguistic analysis
2. Create *urls* attribute that maps application's URL namespace to selected handler. In this case, we have only one handler
3. Optionally, a standalone version of the app can be created e.g. by invoking a built-in method from Wille

Listing 4.3.5.1 *Code skeleton for a simple HTML GUI that handles GET/POST actions*

```
 1 class HTMLKeywords:
 2     def GET(self, request):
 3         return request.render_template('upload.html', dict())
 4 
 5     def POST(self, request):
 6         client = request.wille_client()
 7         # Access Wille2 Services and transform input from request.input()
 8         # Write results into a string and return it
 9         str = ''
10         return str
11 
12 urls = (
13         ("/", HTMLKeywords),
14 )
15 
16 if __name__ == "__main__": wille.app.commandline_app(urls)
```

The resulting App can be used, as described in the previous section, either as a standalone application or as an app in Wille2 server.

5. Scenarios

In order to enable OPAALS consortium to work more efficiently and to focus the development of Wille2 visualisation framework, we implemented a set of visualisation scenarios in parallel to the development of the visualisation system.

Scenarios include among others the following visualisation themes:

- Development of OPAALS 2008 conference workspace.
- Visualisation of different social network configurations of OPAALS wiki and OPAALS 2008 community.
- Visualisation of large heterogenous data sets in order to support visual data mining.
- Visualisation Integration via Embedding and designer/author collaboration

5.1. OPAALS 2008 Workspace

The 2nd International Conference on Open Philosophies for Associative Autopoietic Digital Ecosystems (OPAALS 2008) was organised on 7th and 8th October 2008 in Tampere, Finland by the Hypermedia Laboratory of Tampere University of Technology (TUT) in association with Mindtrek Conference and W3C Finnish Office. As HLab was responsible for organising OPAALS 2008, the development process of the conference workspace was selected as a real-life pilot scenario for the visualisation system. OPAALS 2008 was a relatively small conference with about 50 registered delegates representing fields of, among others, social science, biology, and computer science. Conference scenario is discussed in detail in (Huhtamäki, Nykänen and Salonen, 2009).

The main outputs of the scenario are

1. online OPAALS 2008 Workspace at http://matriisi.ee.tut.fi/hypermedia/events/opaals2008,
2. an offline version of the workspace in CD-ROM format, and
3. a manually edited, more traditional proceedings book *Proceedings of the 2nd International OPAALS Conference on Digital Ecosystems: OPAALS 2008, 7-8 October 2008, Tampere, Finland* (Nykänen, Huhtamäki, Salonen, Pohjolainen and Silius, 2008).

A few efforts in modelling and visualising academic conference data were particularly inspiring for our work:

In order to demonstrate the real-life usage of Semantic Web technologies and to test the workflows and tools needed to collect semantically rich data in a realistic setting, or to eat the "Semantic Web Dog Food", conference metadata about people, papers and talks has been created for a number of European and International Semantic Web Conferences (ESWC and ISWC) (Möller, Heath, Handschuh and Domingue, 2008). The metadata is represented in Resource Description Framework (RDF) format following the ESWC2006 Conference Ontology. The ontology combines concepts of the Friend of a Friend (FOAF) vocabulary to concepts specific to the conference domain. Currently, data representing six academic conferences is available online both as a browsable repository and via an Application Programming Interface (API) supporting SPARQL Query Language for RDF (SPARQL).

Molka-Danielsen, Trier, Slykh, Bobrik and Nurminen (2007) present a particularly inspiring case of visualising the evolution of an academic community. To visualise the evolution of the social network of authors and the research topics of IRIS (Information Systems Research Seminar in Scandinavia) on a period from 1978 to 2006, data representing articles and their authors was collected manually from web pages, conference proceeding CDs and other sources and inserted into a database. A dynamic network visualisation tool Commetrix was used to visualise the data. The outputs of the work include animations of the evolution of IRIS co-authorship and of the diffusion

of terms in IRIS community.

In his winning proposal for the Semantic Conference Design Challenge of the 3rd Annual European Semantic Web Conference, Grimnes (2006) nicely sums up many of the needs of a conference delegate: delegates should be able to efficiently browse conference program and tag talks, on basis of which a conference system, where the user data could possibly be initialised with existing profile information in FOAF format, could guide the delegate intelligently through the conference. Today, such intelligent online communities can be built via integrating and tailoring existing community engines and other Web 2.0 development tools.

The implementation of the conference data processing and workspace creation was done with Wille. As Wille2 was at a pre-alpha stage in the beginning of the workspace development process, the Ant-based Wille1 was used for the task.

5.1.1. Workspace development process

Figure 5.1.1.1 depicts the OPAALS 2008 workspace development process. A detailed description of the process follows.

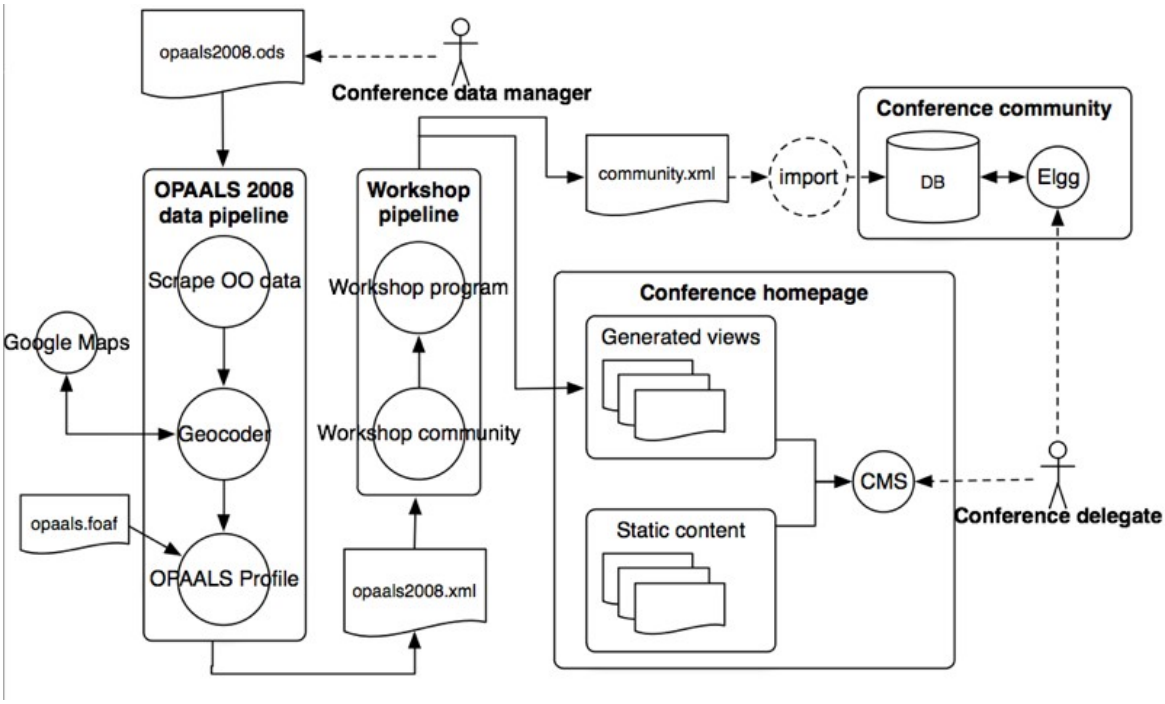


Figure 5.1.1.1 *A diagram presenting the workflow of OPAALS 2008 workspace development*

Traditional conference management tools were used to collect OPAALS 2008 data. Articles were submitted through an online open source conference management system OpenConf that was installed for OPAALS 2008. Conference delegates registered via Mindtrek registration system and the registration data was delivered to the OPAALS 2008 team every now and then via email. Since OpenConf (we used version 2.01) does not provide a separate management process of camera-ready articles, authors submitted the final versions of articles via email in Portable Document Format (PDF). The files were named according to a uniform naming scheme to enable mechanised referencing.

OPAALS 2008 conference data is composed of *article data* (id, author(s), title, session, keywords, abstract, full article in PDF format), *delegate profile information* (id, name, conference community username, email, organisation, location [city, country] and a list of interests) and *session*

information (id, title, type, chair, start time, end time). Moreover, for members of OPAALS, *additional profile information* (image, additional list of interest) was collected from the OKS. For this, OPAALS username was added to the conference data to connect the OPAALS member identities to respective OPAALS 2008 delegate identities.

In order to avoid time-consuming software development, a (human) conference data manager conducted the aggregation of conference data, an effort resulting to an Open Office Calc spreadsheet (opaals2008.ods in Figure 5.1.1.1). A few iterations were made to collect sufficient data that fully supported the creation of the needed visualisations.

Although a human actor was used, efforts were made to smoothen the process. For exporting article information from the submission management system via a Web interface, a data API capable of accessing the OpenConf database and exporting the data in a machine-readable format was implemented. Moreover, a set of Wille2 pipelines were developed for processing the data: Two previously developed pipelines, one for getting OPAALS member data in canonical Friend of a Friend (FOAF) format (opaals.foaf in Figure 5.1.1.1) (see Huhtamäki, 2007) and another for fetching OPAALS member images were re-used.

Listing 5.1.1.1 *Example of OPAALS member data in FOAF*

```
   <foaf:Person rdf:about="mailto:huhtis@opaals.org">
      <foaf:img rdf:resource="http://wiki.opaals.org/huhtis/FactsAboutPerson?
action=AttachFile&amp;do=get&amp;target=opaals_fact_person_image.jpg"/>
      <foaf:name>Jukka Huhtamäki</foaf:name>
      <foaf:mbox rdf:resource="mailto:jukka.huhtamaki@tut.fi"/>
      <foaf:Organization rdf:resource="http://wiki.opaals.org/TUT"/>
      <foaf:interest>
         <foaf:Document rdf:about="http://wiki.opaals.org/Information Visualisation">
            <dc:title>Information Visualisation</dc:title>
            <dc:description>Existing wiki page</dc:description>
            <dc:identifier rdf:resource="http://wiki.opaals.org/Information
Visualisation"/>
         </foaf:Document>
      </foaf:interest>
      <foaf:interest>
         <foaf:Document rdf:about="http://wiki.opaals.org/Social_networking">
            <dc:title>Social networking</dc:title>
            <dc:identifier rdf:resource="http://wiki.opaals.org/Social_networking"/>
            <dc:description>A concept</dc:description>
         </foaf:Document>
      </foaf:interest>
      <!-- Interest list is truncated: see Appendix for a full version →
      <foaf:homepage rdf:resource="http://matriisi.ee.tut.fi/hypermedia/"/>
      <foaf:publications rdf:resource="http://trip.cc.tut.fi/julkaisut/english.html"/>
      <foaf:holdsAccount>
         <foaf:OnlineAccount>
            <rdf:type rdf:resource="http://opaals.org/draft/workspace/wiki"/>
            <foaf:accountServiceHomepage rdf:resource="http://wiki.opaals.org/"/>
            <foaf:accountName>huhtis</foaf:accountName>
         </foaf:OnlineAccount>
      </foaf:holdsAccount>
   </foaf:Person>
```

In addition to reusing the existing OPAALS FOAF pipeline, a dedicated OPAALS 2008 data pipeline was developed for picking the spreadsheet content from the Open Office Calc file (which is, effectively, a zip file including spreadsheet content in XML format), scraping the data load, geocoding delegate origins with Google Maps API, adding pre-fetched OPAALS member profile information and, finally, creating an Extensible Markup Language (XML) representation of the conference data (document opaals.xml in Figure 5.1.1.1).

Listing 5.1.1.2 *Example of OPAALS 2008 delegate data*

```
    <delegate id="huhtamaki">
        <firstname>Jukka</firstname>
        <lastname>Huhtamäki</lastname>
        <email>jukka.huhtamaki@tut.fi</email>
        <organisation>
            <name>Hypermedia Laboratory of Tampere University of Technology</name>
        </organisation>
        <city>Tampere</city>
        <country>Finland</country>
        <address xmlns:kml="http://earth.google.com/kml/2.0" type="google.geoinfo">
            <name>Tampere, Finland</name>
            <lat>61.4980229</lat>
            <lng>23.7648591</lng>
            <!-- Location details are truncated: see Appendix for a full version →
        </address>
        <image uri="http://wiki.opaals.org/huhtis/FactsAboutPerson?
action=AttachFile&amp;do=get&amp;target=opaals_fact_person_image.jpg"/>
        <!-- List of interests from the MindTrek registration form →
        <interestlist>
            <interest>community networks</interest>
            <interest>community visualisation</interest>
            <interest>data processing pipelines</interest>
            <interest>digital ecosystems</interest>
            <interest>social network analysis</interest>
            <interest>socio-technical perspective</interest>
        </interestlist>
        <opaals:profile accountname="huhtis">
           <!-- OPAALS profile of a delegate is located here, see Listing 5.1.1.1 →
        </opaals:profile>
    </delegate>
```

An example of article data is represented in Listing 5.1.1.3.

Listing 5.1.1.3*. Example of article information*

```
    <article id="opaals2008-article14" type="article">
         <title>A Visualisation System for a Peer-to-Peer Information Space</title>
         <abstract>Peer-to-peer (P2P) networks provide new opportunities and
challenges for distributing and using data and services without a centralised design.
An interesting use case for P2P systems is the interplay of information repository and
data visualisation applications. In this article, we investigate the potential of
distributed information and data processing services and evaluate their applicability
in visualisation systems. ... / ... Finally, we demonstrate the technical
implementation via pre-studies and discuss P2P knowledge as a complementary topic for
P2P visualisations. The work is based on an ongoing research project that will deliver
a component-based P2P visualisation system, in a European network of excellence
thriving to establish a distributed open knowledge system as a digital ecosystem,
based on the ideology of open source communities.</abstract>
      <!--Note: the id-attribute of each author references to the delegate list.-->
      <authorlist>
            <author id="nykanen"/>
            <author id="salonen"/>
            <author id="haapaniemi"/>
            <author id="huhtamaki"/>
        </authorlist>
        <slides format="pdf"/>
        <keywordlist>
           <keyword>visualisation</keyword>
           <keyword>peer-to-peer networks</keyword>
           <keyword>pipeline processing systems</keyword>
           <keyword>digital ecosystems</keyword>
        </keywordlist>
      </article>
```

Also program information is represented in XML. See Listing 5.1.1.4 for a detailed example.

Listing 5.1.1.4 *Excerpt of program information*

```
<programitem type="session" id="session1" communityref="session1">
     <start>2008-10-07T10:30:00</start>
     <end show="no">2008-10-07T16:00:00</end>
     <title>Session 1: Knowledge, collaboration, and governance</title>
     <presentationlist>
        <presentation articleref="opaals2008-article22">
           <presenterlist>
              <presenter id="bharadwaj"/>
              <presenter id="sarkar"/>
           </presenterlist>
        </presentation>
        <presentation articleref="opaals2008-article20">
           <presenterlist>
              <presenter id="brauer"/>
              <presenter id="crone"/>
              <presenter id="durrenberg"/>
              <presenter id="lapteva"/>
              <presenter id="zeller"/>
           </presenterlist>
        </presentation>
        <presentation articleref="opaals2008-article18">
           <presenterlist>
              <presenter id="brauer"/>
              <presenter id="crone"/>
              <presenter id="durrenberg"/>
              <presenter id="zeller"/>
           </presenterlist>
        </presentation>
     </presentationlist>
  </programitem>
```

5.1.2. Conference Workspace Introduction

The OPAALS 2008 conference workspace is designed to support conference delegates to (1) prepare for the conference, (2) work effectively during the conference and (3) follow-up discussion and initiatives after the physical event. The workspace is two-folded: a traditional conference homepage is spiced up with conference data visualisations and complemented with a dedicated conference community driven by an open source social network engine Elgg .

The following visualisations are included in the workspace. *Interactive program* (see Figure 5.1.2.1) is a densely hyperlinked view to the conference program. *Presenter list* introduces all the presenters appearing in OPAALS 2008. *Article list* enumerates all the articles published in OPAALS 2008 proceedings. The full version of each article is linked to Article list in PDF as well as the related conference presentations that were delivered to the OPAALS 2008 team. A *keyword cloud* (see Figure 5.1.2.2), a tag cloud (see e.g. Hearst and Rosner, 2008) based on a histogram of conference article keywords is provided to support browsing the articles by their subject. Finally, *Presenter map* (see Figure 5.1.2.3), a traditional map-based mash-up utilising Google Maps, shows the whereabouts of each presenter giving a talk in OPAALS 2008.

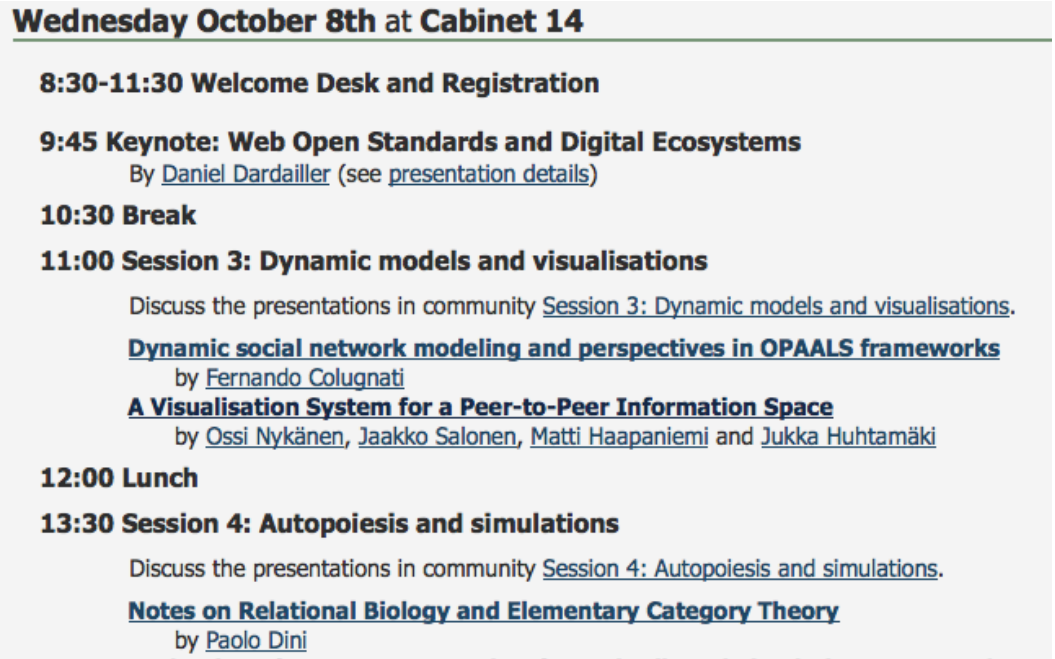


Figure 5.1.2.1 *Excerpt of OPAALS 2008 Interactive program*

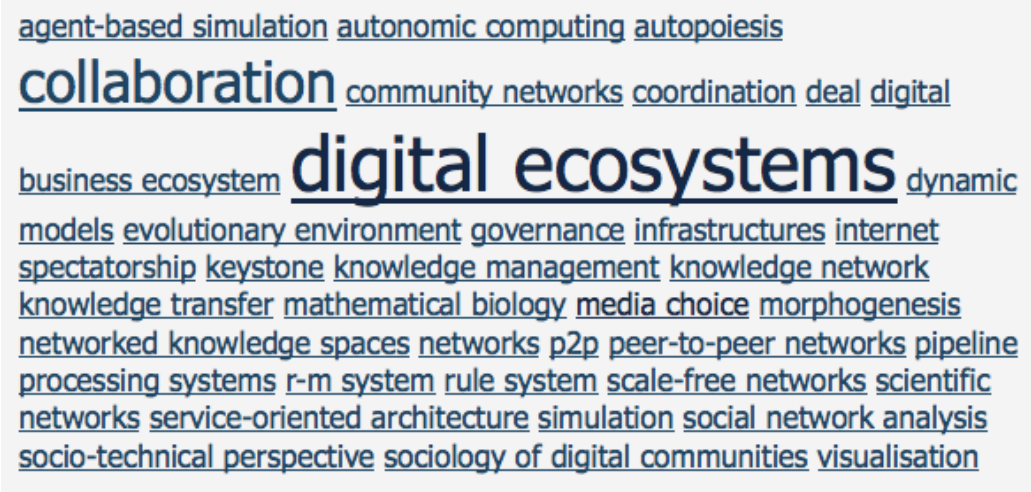


Figure 5.1.2.2 *OPAALS 2008 keyword cloud*

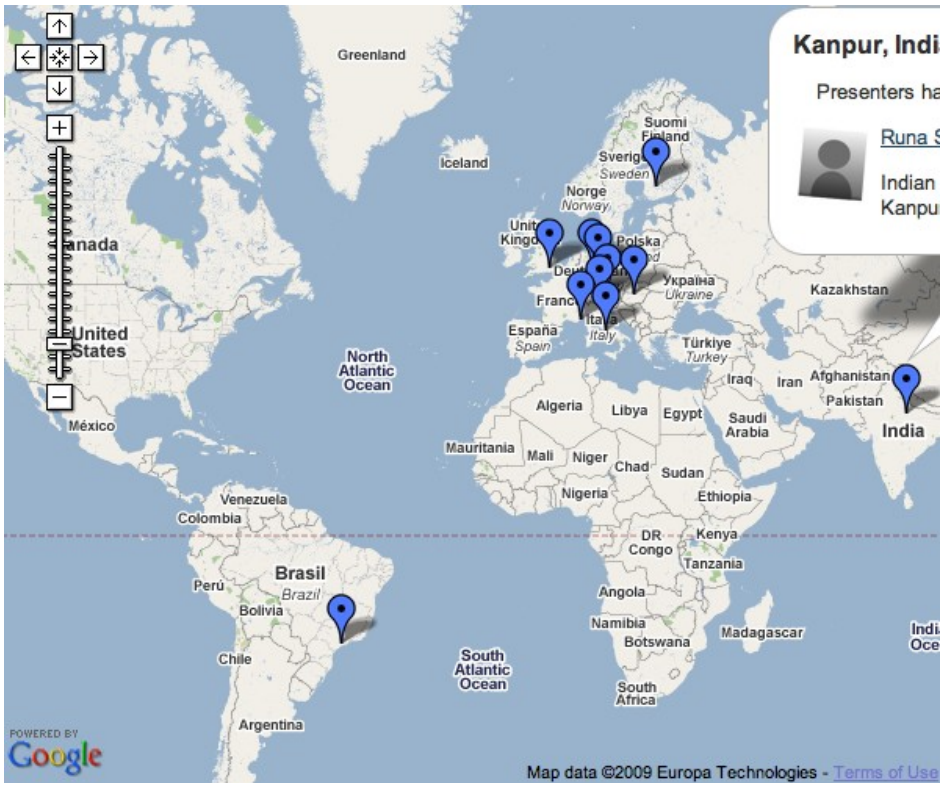


Figure 5.1.2.3 *OPAALS 2008 Presenter map*

The interlinking of the different views of the workspace is tight, thus providing flexible means to browse the conference contents. Moreover, conference sessions are linked to discussions running in the conference community and author information to their user profiles in the community. A simple Content Management System (CMS) was built to serve the static content as well as the generated views within a page template providing a uniform navigation scheme and general look and feel to the conference homepage. The CMS does not implement user session management or authentication, thus creating e.g. a personal tagging system for program was not possible.

A general Workshop pipeline was built to produce the needed visualisations and population data files. The pipeline is composed of components Workshop program and Workshop community, first creating the Interactive program, the Delegate list, the Article list and marker data for the Presenter list (markers represent information in a Google Map) and second creating a data file (community.xml in Figure 5.1.1.1) used to populate the conference community. A set of tailored XML vocabularies are used to represent the data within the pipelines. Most of the data processing is implemented as Extensible Stylesheet Language Transformations (XSLT). The visualisations are based simply on Hypertext Markup Language (HTML) and Cascading Stylesheets (CSS). In addition, Presenter Map uses Google Maps basic technologies, Keyhole Markup Language (KML) and the Javascript API of Google Maps.

A mildly tailored version of Elgg (version 0.9) was used to run the conference community. A bash script was implemented to create a community account for each delegate, to populate the community profiles with collected data and to deliver the account information to delegates via email. To minimise the efforts of delegates and to enable linking from the Presenter list to the conference community, the conference data manager defined the community usernames manually into the conference data.

A stand-alone version of the workspace was created through parameterisation of the Workshop pipeline and disseminated as a CD-ROM. Finally, a manually edited version of OPAALS 2008 proceedings in PDF format was included both into the conference homepage and the CD-ROM.

During the development of the OPAALS 2008 workspace, feedback from different visualisations and features of the conference workspace was collected from the members of the OPAALS 2008 team in face-to-face meetings and, in addition, from social network usefulness experts through informal discussions and reviews. Moreover, in order to provide the workspace development team with concrete data on the workspace usage before, during and after the conference, we installed Google Analytics, an online usage tracking and visualisation tool, for collecting and visualising the usage data. The development of workspace visitor count in Figure 5.1.2.4 presents an example of a view provided by Google Analytics. To get more precise information about the user behaviour, a tailored log collector was implemented to the conference community platform by utilising the extension mechanism of Elgg.

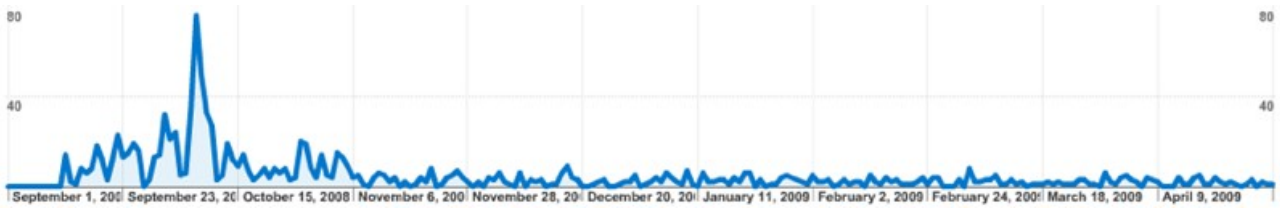


Figure 5.1.2.4 *OPAALS 2008 visitor count view in Google Analytics*

Google Analytics enables us to gain insight on the usage of the workspace in a general level. The information derived from usage tracking and visualisation can be used to reflect the dynamics of the workspace. For instance, we decided to separate the traditional, static homepage of the conference from the more interactive workspace. This meant that we had two versions of the conference program, one with hyperlinks to different parts of the workspace and another without. In the period of September 15, 2008 to February 15, 2009, the unlinked version of the program was viewed twice as much as the interactive version. We suspect that this may have had a major effect on the popularity of the other parts of the workspace. In the future, we would create only one program that would initially be a static one and would be replaced by the interactive version once enough data for creating one is available.

Moreover, on basis of the community log information, we are e.g. able to generate a visualisation of the evolution of the social structure of the community. The analysis of the evolution of OPAALS 2008 social network is discussed in more detail in Chapter 5.2.

5.2. Social Network Analysis and Visualisation

Social network visualisations are, in general, very useful for investigators of social networks in providing "new insights about network structures" and help "them to communicate those insights to others" (Freeman, 2000). Visualising the social network of OPAALS wiki serves both researchers of as well as the members of OPAALS using the wiki (cf. e.g. (Huhtamäki, 2007)).

We have focused our visualisation efforts in showing the social configurations of OPAALS wiki contributors. However, in order to ensure a more general approach, the created SNA tools were also used to analyse and visualise OPAALS 2008 workspace usage data as well as other data sets not related to OPAALS.

Moreover, since there is a variety of tools that can be used to analyse and visualise social network data, we are using a set of general data formats in representing social network data enabling us to support different low-level data formats supported by individual tools. Tools for analysing social network data include, among others, Pajek (http://pajek.imfm.si/), Vizster (http://jheer.org/vizster/), Orange (http://www.ailab.si/Orange/), R (http://www.r-project.org/), and SPSS (http://www.spss.com/) as well as traditional spreadsheet tools.

5.2.1. OPAALS wiki contributor network

OPAALS wiki contributor network is a command-line tool for collecting and processing data on the contribution history of OPAALS wiki to support social network analysis and visualisation. In practice, the tool is a Python file `opaals-wiki-sna.py` located in `apps/opaals-wiki-sna` folder in the Wille2 bundle. More specifically, it follows the Processor architectural configuration (see Chapter 4.1). The tool is implemented in a way that a GUI can be implemented if seen appropriate.

Currently, tool supports the following options for collecting the pages:

- All pages in the wiki: `*`
- Recent changes in wiki pages: `recent:1|2|3|7|14|30|60|90`
- One or more page categories: `CategoryFirst,CategorySecond,CategorySecond/Subcategory`

The tool is run in the following manner:

```
> python opaals-wiki-sna.py recent:30 username password
```

When the example process is launced, all pages that have been edited during the last 30 days are fetched and a full contribution history is created on basis of the data.

In OPAALS wiki, users contribute by editing individual wiki pages. As wikis, by definition, manage the editing history of individual pages, wikis have a built-in contribution tracking mechanism. For example, the contributions of wiki page "OKS - Feature List" (http://wiki.opaals.org/OKSSpecificRequirementsGathering) are listed in Revision History view of the page (http://wiki.opaals.org/OKSSpecificRequirementsGathering?action=info). For each contribution, date, size after, contríbutor and the type of contribution (edit, adding or removing an attachment) can be distilled from the revision history.

While a more high-level query API for OKS wiki is absent, we are using the default HTTP API for accessing the wiki contribution history data. Following Wille2 components are used to carry out the data collection process:

- **opaals-wiki-search**: Lists wiki pages by category, latest edited pages or a full list of pages. The component can be extended to support keyword search or other means of query as needed.
- **resourcefetcher**: Authenticates to OKS wiki by using the given credentials and fetches a specified set of resources.
- **tidy**: Fixes charset and other errors in HTML and XML to enable processing with tools that require valid XHTML/XML.
- **wiki.pagehistory**: Scrapes contribution history from an HTML-formatted page. Requires valid XHTML.
- **wiki.historyeventlog**: Creates a log representation on basis of a wiki history in XML.
- **sna.log.graph**: Creates a graph representation of a social network on basis of a logfile.

In detail, the visualisation process operates in the following manner:

1. List pages: all pages or one or more categories.
2. For each page:
    1. Fetch a page from the OKS wiki.
    2. Make page a valid one.

    3. Scrape page editing history.
    4. Aggregate the results to the full editing history.
3. Create a log file from editing history.
4. Create a Pajek-formatted file from log file.

A rationale for introducing a step of producing a log file from the wiki editing history is given in the following chapter.

It is obvious that the data collection process based on HTTP and particularly HTML is very easily breakable until a more high-level data access will be provided. On the other hand, a working social network visualisation implementation demonstrates that we are able to visualise also heterogenous sources of data.

An example of the HTML-formatted editing history of a wiki page:

```
    <tr>
        <td>68</td>
        <td>2008-12-18 12:15:32</td>
        <td>17172</td>
        <td><input type="radio" name="rev1" value="68"><input type="radio" name="rev2"
            value="68"></td>
        <td><span title="tkurz @ 193.171.53.131[193.171.53.131]"><a href="/tkurz"
            title="tkurz @ 193.171.53.131[193.171.53.131]">tkurz</a></span></td>
        <td> </td>
        <td> <a href="/OKSSpecificRequirementsGathering?
action=recall&amp;rev=68">view</a> <a href="/OKSSpecificRequirementsGathering?
action=raw&amp;rev=68">raw</a> <a href="/OKSSpecificRequirementsGathering?
action=print&amp;rev=68">print</a></td>
    </tr>
```

After the HTML markup is made valid (with a Wille2 component providing a HTML Tidy service), it can be processed to create an XML presentation of the data:

```
    <history pageid="http://wiki.opaals.org/OKSSpecificRequirementsGathering">
        <contribution contributorid="tkurz" date="2008-12-18 12:17:17" revision="69"
 size="16642" type="edit"/>
        <contribution contributorid="tkurz" date="2008-12-18 12:15:32" revision="68"
 size="17172" type="edit"/>
        <contribution contributorid="tvprabhakar" date="2008-12-18 11:03:20"
revision="67" size="17341" type="edit"/>
        <!-- The example is truncated →
    </history>
```

In general, a log file can be generated to represent the actions done in a system. Thus, the log file approach provides a general way to analyse the activity of a given social network. Also wiki contribution history can be expressed as individual log events where source (or actor) is the contributor and target is the wikipage:

```
    <eventlist>
        <!-- Example event →
        <event time="2008-12-18 12:17:17">
            <source>tkurz</source>
            <target>http://wiki.opaals.org/OKSSpecificRequirementsGathering</target>
            <type>edit</type>
        </event>
        <!-- The example is truncated →
    </eventlist>
```

Next, the data can be processed to create e.g. a Pajek-compliant representation of the contributor network:

```
    *Network From  to
    *Vertices 68
    8 contributor_tkurz 0.0000 0.0000 0.5000  ic Red bc Black  x_fact 13.15 y_fact
13.15 0.0000 0.0000 0.5000
    ...
    46 wikipage_http://wiki.opaals.org/OKSSpecificRequirementsGathering 0.0000 0.0000
0.5000  ic Green bc Black  x_fact 11.35 y_fact 11.35 0.0000 0.0000 0.5000
    ...
    8 46 35
```

The size of a change of a given edit can be calculated from the editing history information and can thus be taken into account. The quality of the contribution is, however, difficult or impossible to analyse with quantitative means. For now, we treat each edit with an equal value. Parametrisation can, however, be easily introduced as needed.

An example of the wiki contributor network is presented in Figure 5.2.1.1. In addition to *two-mode* (or bi-partite) networks shown in the example, Wille2 SNA tools can be used to produce *one-mode* networks that are more traditional in SNA research.

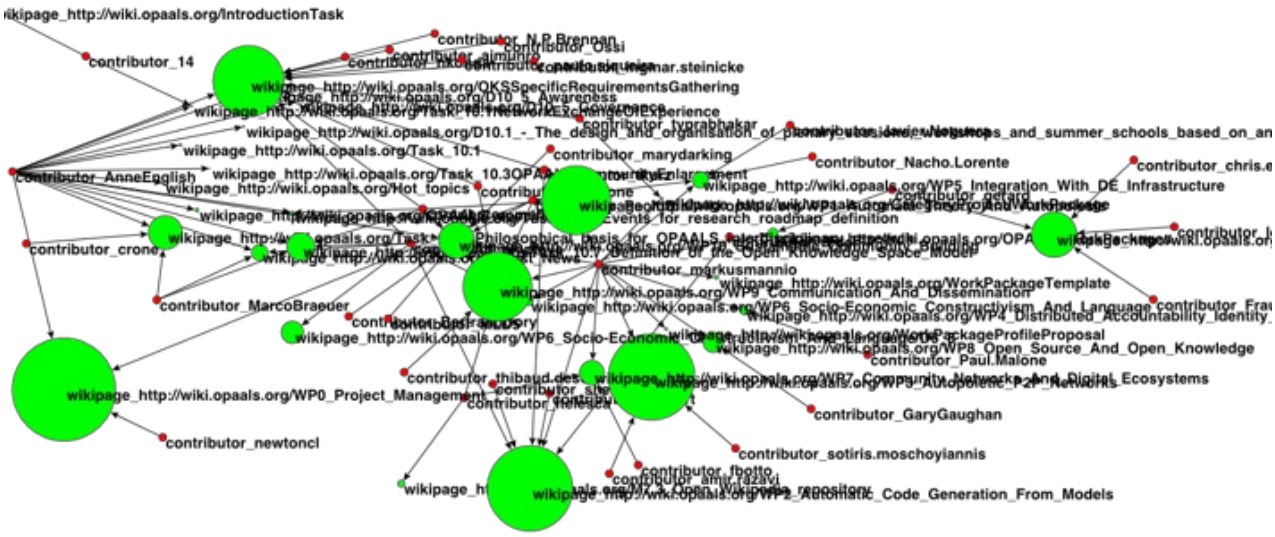
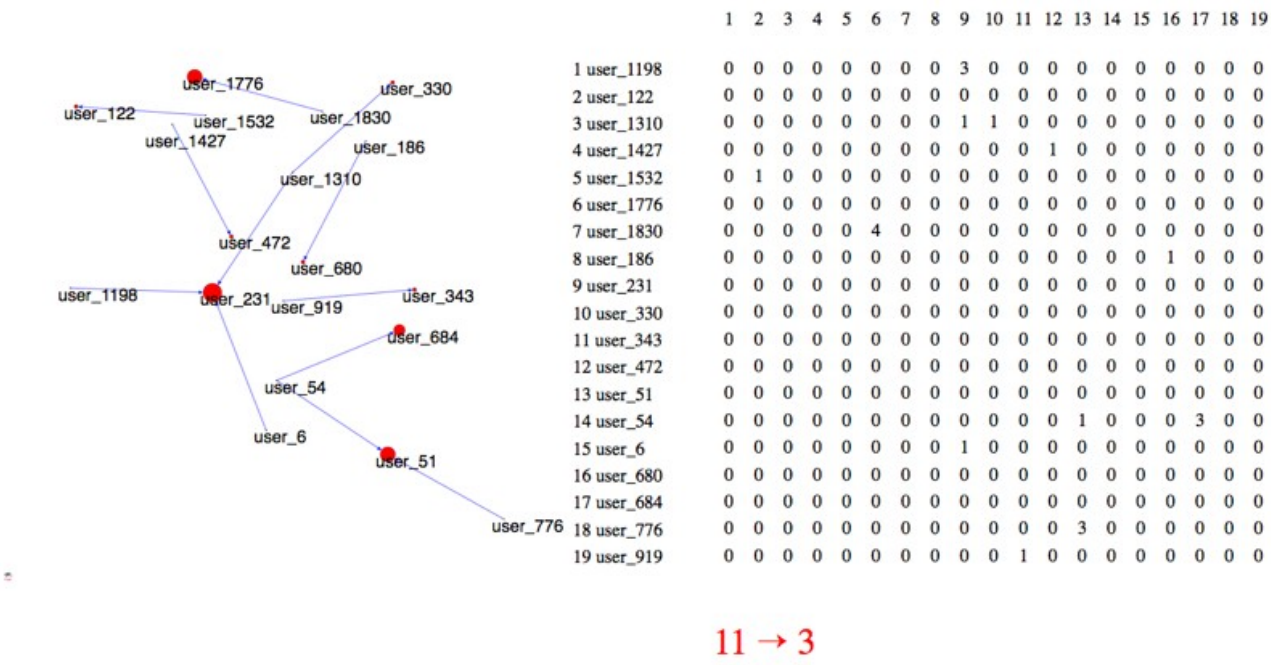


Figure 5.2.1.1 *Example of wiki contributor network visualisation*

The needs posed to SNA data vary dependent whether the data are used for visualisation or analysis. In order to serve both purposes, we have developed a set of serialisation methods for SNA data. For example, a serialisation that can be imported to MATLAB or Octave can be created:

```
    X(1,1)=0;
    X(1,2)=0;
    X(1,3)=0;
    X(1,4)=0;
    X(1,5)=0;
    X(1,6)=0;
    X(1,7)=0;
    X(1,8)=0;
    X(1,9)=3;
```

With the data sets imported to MATLAB, we are able to calculate the centrality and the prestige of different actors in a given social network. When the results of the mathematical analysis are combined with the visualisation of social network, we are able to support the work of social network analysers by generating new kinds of reports. An example of a prototype report is depicted in Figure 5.2.1.2.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1 user_1198	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0
2 user_122	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3 user_1310	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
4 user_1427	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
5 user_1532	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6 user_1776	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7 user_1830	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0
8 user_186	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
9 user_231	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10 user_330	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11 user_343	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12 user_472	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13 user_51	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14 user_54	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	3	0	0
15 user_6	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
16 user_680	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17 user_684	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18 user_776	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0
19 user_919	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

11 → 3

Evaluated for dichotomous relations

Index	Actor degree centrality	Actor degree prestige	Actor proximity prestige
1	0,0556	0	0
2	0	0,0556	0,0556
3	0,1111	0	0
4	0,0556	0	0
5	0,0556	0	0
6	0	0.0556	0.0556

Figure 5.2.1.2 Example view of an interactive SNA report prototype

A set of human-made efforts is still needed in order to create the graph representation of the social network configuration as well as the animation of it's evolution. In the future, we will investigate means to create automated components that are able to create both static views and animations of a social network.

5.2.2. OPAALS 2008 community evolution

Another social network emerged related to OPAALS 2008 conference where an online community was provided for conference delegates. A tailored mechanism was implemented to track usage data and to enable exporting of the data in XML format for processing. Usage data from OPAALS 2008 community can be exported as a log file:

```
<event time="2009-01-21T10:24:51">
    <source>3</source>
    <target>34</target>
    <act>FORUMPOST</act>
</event>
```

An example visualisation of a social network is depicted in Figure 5.2.2.1. Nodes represent the community members and the theme groups in the conference community and lines represent the interconnections between them. The graph representation of the community is created with a social network analysis tool Pajek. In OPAALS 2008, the profile views constitute the majority of the community activity, thus making the social network data less significant. Similar means can, however, be used to create social network visualisations in conferences where online interaction is more diverse. Moreover, animations of social network evolution can be created from the log data e.g. with SoNIa, a tool for visualising dynamic, longitudinal network data.

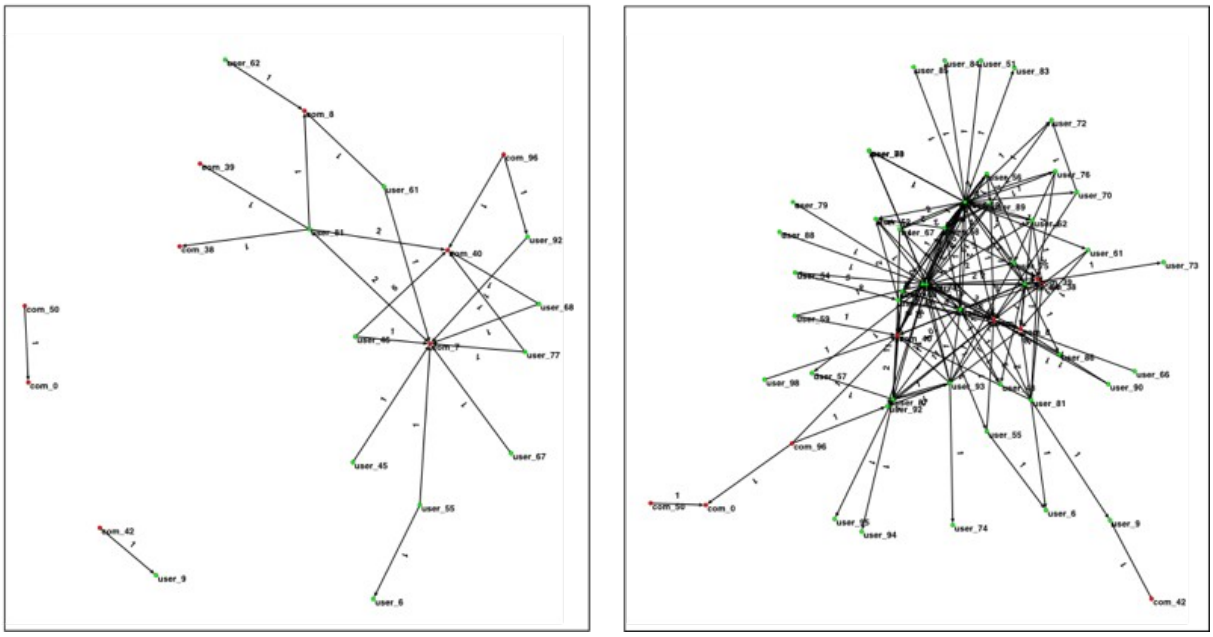


Figure 5.2.2.1 *Evolution of the conference delegate social network (two frames)*

In order to visualise the configuration of a social network in a given time period, a possibility to select a slice of data on basis of time is important for the flexibility of analysis. In Google Analytics, for example, an analyst of the behaviour of Web site usage is able to easily define a specific period of time to be examined. With Wille2 SNA-tools, a timeslice can be defined by specifying start and end dates, e.g. `2008-06-01`.

In order to visualise the evolution of a community, for example, we need a set of consecutive slices of time. Social network visualisation and analysis tools such as Commetrix and SoNIa enable the visualisation of evolution. While it is currently not possible to fully automate the creation of such a visualisation, we are able to fine-tune a set of data by hand and create e.g. a visualisation of the development of OPAALS 2008 community.

5.3. Explorative Visualisations with Data Mining and Visual Programming

Intuitively, a visualisation illustrates a phenomenon that is implicitly present in the source dataset. However, in practice, the raw data is seldom (by a change) in a form that directly yields a visualisation. Rather, the emergent properties of data must be first explicitly computed into a suitable form, before the mapping onto a visualisation model can take place. In many cases, this may require using the techniques of, e.g., statistical analysis, data mining, and machine learning.

5.3.1. Data Mining for Data Transformations

In the abstract sense, complexity in designing visualisations is due to the two conceptual steps in the abstract visualisation pipeline: data transformations and visualisation transformations. When data is presented with respect to a general-purpose visualisation model (such as data structure capturing a graph or a discrete three-dimensional plot), the role of data transformations is emphasised.

From this perspective, we may refine the model of a visualisation pipeline by explicitly recognising the data transformation steps as follows:

1. Data Access
2. Selection and Pre-Processing
3. Itemisation
4. Computations with Items
5. Mapping Results onto a Visualisation Model

Of course, the whole process starts from data access which provides raw data to work with. However, for data quality and capacity reasons, raw data must typically be filtered and harmonised, so that it can be conveniently manipulated. Itemisation refers to the activity of organising data into objects with properties (items). Once a suitable item representation of the data is computed, it is mapped onto a visualisation model. This mapping typically includes projections from item properties onto the spatial or topological properties of a visualisation model.

While itemisation and item computations could be considered as an implicit part of either the pre-processing or the mapping activities, they are significant both conceptually and in practice. First, itemisation provides application-specific conceptualisation of the data. Second, practical algorithms are implemented to be worked with particular data structures.

A textbook example of itemisation is encoding data in a tabular form (i.e. rows represent items -- or instances -- and columns represent item properties -- or attributes). The virtue of this kind of itemisation lies in the available repertoire of algorithmic tools in manipulating and computing values from this data. However, while tabular data form is clearly useful for basic data transformations (e.g., matrix algorithms), it is not always suitable for irregular, semantically extensible data or visualisation formats where objects may include complex internal structure and references. Instead, e.g. the structure of document fragments may be more natural.

This highlights the importance of itemisation: By appropriate conceptualisation and formatting data, several data mining etc. techniques can be directly introduced into the abstract data transformation process. Further, because of the limitations of relational data (highly regular structure), transformations in general involve mappings between tabular and other forms, most notably (fragments of) structured documents.

5.3.2. Host Application Integration using Orange Canvas

In the case of visualisations that include complex data transformations, the toolkit of a visualisation developer should include the basic statistical and data mining techniques. Further, while scripting in principle provides a sufficient setting for designing applications, many designers at some point appreciate visual development tools, including, e.g., file management, graphical pipeline editing, and widgets of dialog-based configuration of the (data) transformation tasks. It turns out that both of these concerns can be addressed by choosing an appropriate host application for the visualisation system.

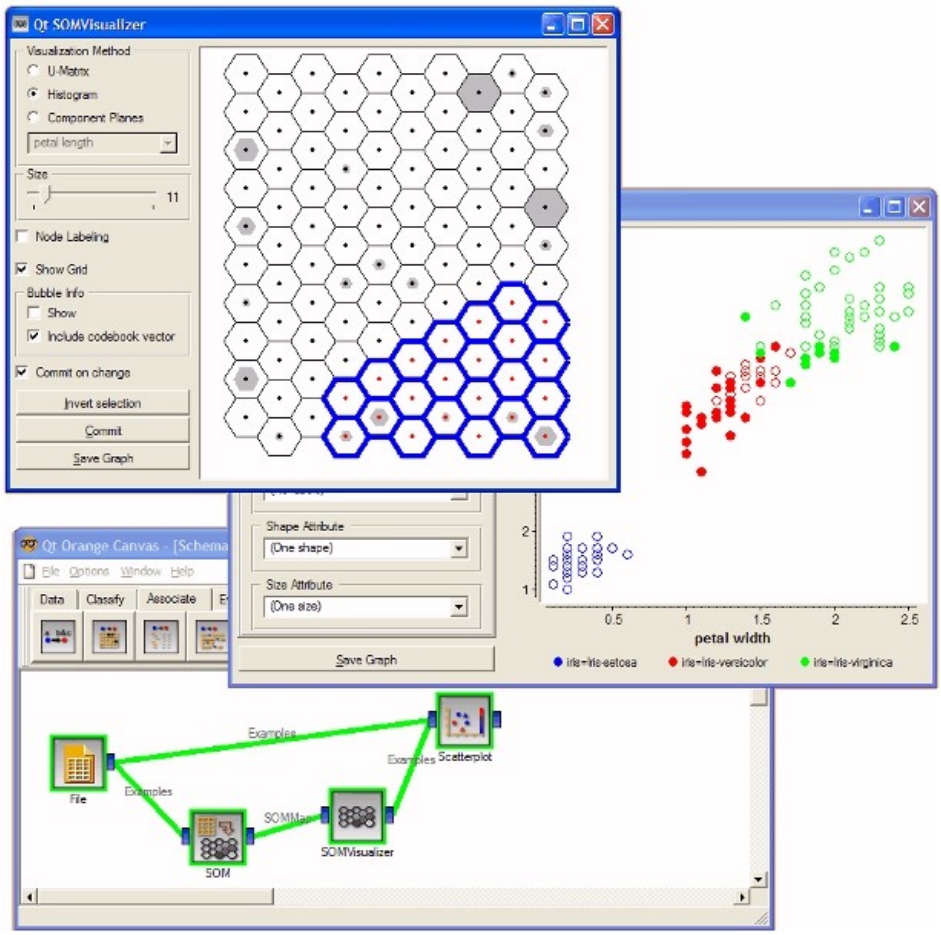


Figure 5.3.2.1. *Orange Canvas Graphical User Interface (see http://www.ailab.si/orange/screenshots.asp)*

A very nice 3rd-party visual programming application, called Orange Canvas, provides an extensible pipeline-based system for developing data mining applications in based on the Orange library in Python (see http://www.ailab.si/orange). In short, the Orange Canvas enables designing, archiving, and executing complex data mining and visualisation pipelines through a graphical user interface (see Figure 5.3.2.1).

The Orange Canvas implements pipeline tasks as Python widgets. Since registering new widgets to the Orange Canvas is possible, we may effectively introduce the appropriate Wille2 functionality in terms of a set of Wille2 widgets. This leads into a refined view to the abstract visualisation system architecture (see Figure 5.3.2.2).

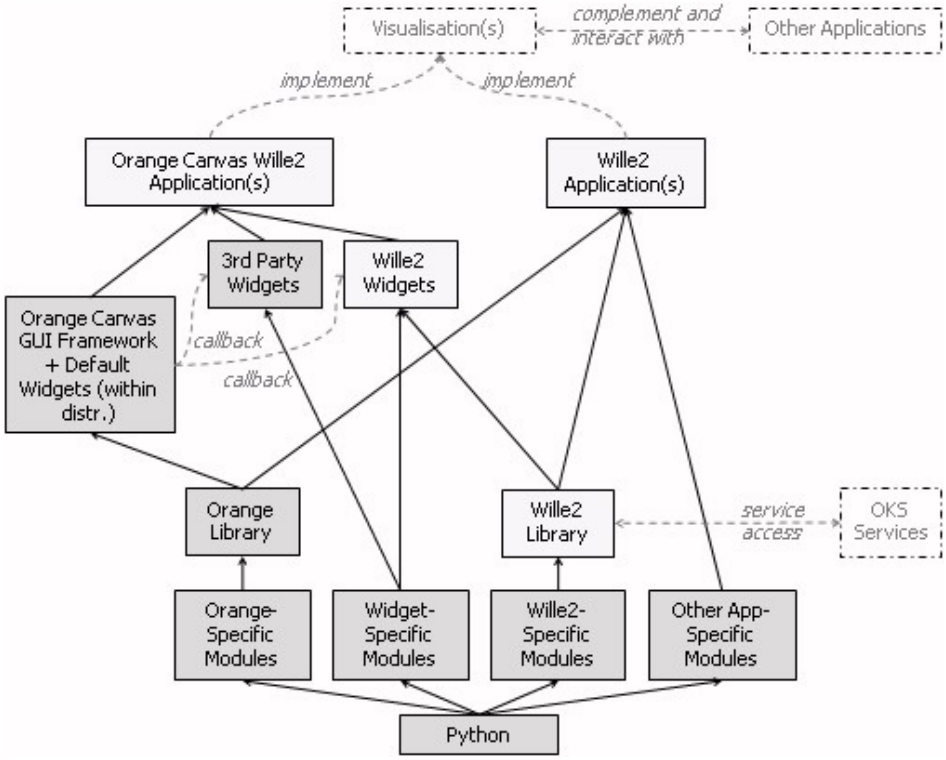


Figure 5.3.2.2 *Abstract visualisation system layer architecture when using Orange Canvas as a host application*

Orange is distributed free under GPL (both binary and source). It is available for Windows, Linux, and Mac OS. Additional libraries exist. (See the online catalog of Orange widgets at http://www.ailab.si/orange/doc/widgets/catalog/.)

Figure 5.3.2.3 *Wille2 widgets for Orange Canvas*

For purposes of Orange integration, the following Wille2 Orange widgets were developed (see Figure 5.3.2.3 above):

- Script Widget: General-purpose Python scripting
- URLRead Widget: Accessing Web resources
- XSLT Widget: Executing text format transformations
- ZipblobMerge Widget: Viewing and integrating data objects
- Browser Widget: Launching a Web browser

The number of the so far implemented widgets is relatively small -- all Wille2 functionality is not provided in terms of a specialised widget interface. However, the script widget allows introducing arbitrary computations within pipelines and thus allows exploiting Wille2 functionality. As suspected, new widgets (with specialised interface) may be developed when necessary. (Working with Wille2 Orange Widgets is more closely discussed in the Appendix A.)

The process of itemisation is clearly present in the Orange integration. In particular, typical data transformations exploit the built-in Orange widgets for various computations by first transforming data into some Orange datatypes (e.g. orange.Example`Table). This makes several data mining

techniques directly accessible, easing the process of developing visualisations.

The main scope of Orange Canvas is exploring data and visualisations by incremental and visual means. However, since pipelines can be saved as files (called schemas in the Orange Canvas terminology) and visualisations may publish light-weight services, visualisation designers can use the system also collaboratively (e.g. accessing others' data, services, pipelines, and visualisations). When reliable scalability and sharing is required, the services are wrapped and deployed according the general policy of publishing service in the P2P OKS framework.

However, deploying pipelines created with the Orange Canvas, as large-scale OKS services, requires manual refactoring. The inevitable extra effort can be minimised by pre-planning: While the current visual programming application does not allow exporting complete pipelines as functional scripts outside the hosting Orange Canvas, most of the pipeline data transformation functionality can be accessed on the level of the Orange and Wille2 libraries (see Figure 5.3.2.2).

5.3.3. Simple Application Example

The chief contribution of this scenario lies in integrating Wille2 visualisation framework with a visual programming environment for data mining and analysis. However, it is instructional to also consider a simple application example.

In the scenarios presented earlier, considerable work has been done in establishing the required technical Wille2 framework and demonstrating the process of extracting wiki article and author data. Let us next demonstrate enriching and using this data in a simple general-purpose data mining activity.

Consider the simple following data-processing and visualisation pipeline:

1. Retrieving wiki.opaals.org wiki data (consisting of 860 pages of articles, at the time of writing).
2. Itemising wiki data (e.g. a database of pages, each associated with the following attributes: URI, editor, last_edit_date, page_data).
3. Computing word histogram vectors for each page (URI as a key), encoding the number of specific terms appearing each page. (For this particular experiment, a simple terminology adopted from the OPAALS 2008 conference article topics was used; see Figure 5.1.2.2)
4. Computing a distance matrix for the page word histogram vectors.
5. Computing a two-dimensional projection of the pages, using Multidimensional Scaling (MDS).
6. Interactively analysing the data in tabular and graphical formats (e.g. zooming into the planar representation, and filtering data by attributes).

As suspected, these tasks nicely fit into the scope of Wille2 and Orange processing. Figure 5.3.3.1 (below) represents a simple visual Wille2/Orange program, implementing the above processing.

In the above figure, pre-processed wikidata is first read from a proxy server (perhaps published as a Wille2 service). Itemising and computing word histograms are essentially implemented as scripts. Then the data is packed into a Zipblob Data Object, suitable for converting into an Orange built-in datatype, ExampleTable. Once data is in this format, the built-in orange widgets (Data Table, Example Distance, and MDS) can be used for implementing the desired views with very little efforts --- the main requirement is understanding data mining. Multidimensional scaling is a technique which finds a low-dimensional (in our case a two-dimensional) projection of points, where it tries to fit the given distances between points as well is possible. Using the Orange MDS widget, it is now possible to easily parametrise both the MDS algorithm and the related visualisation.

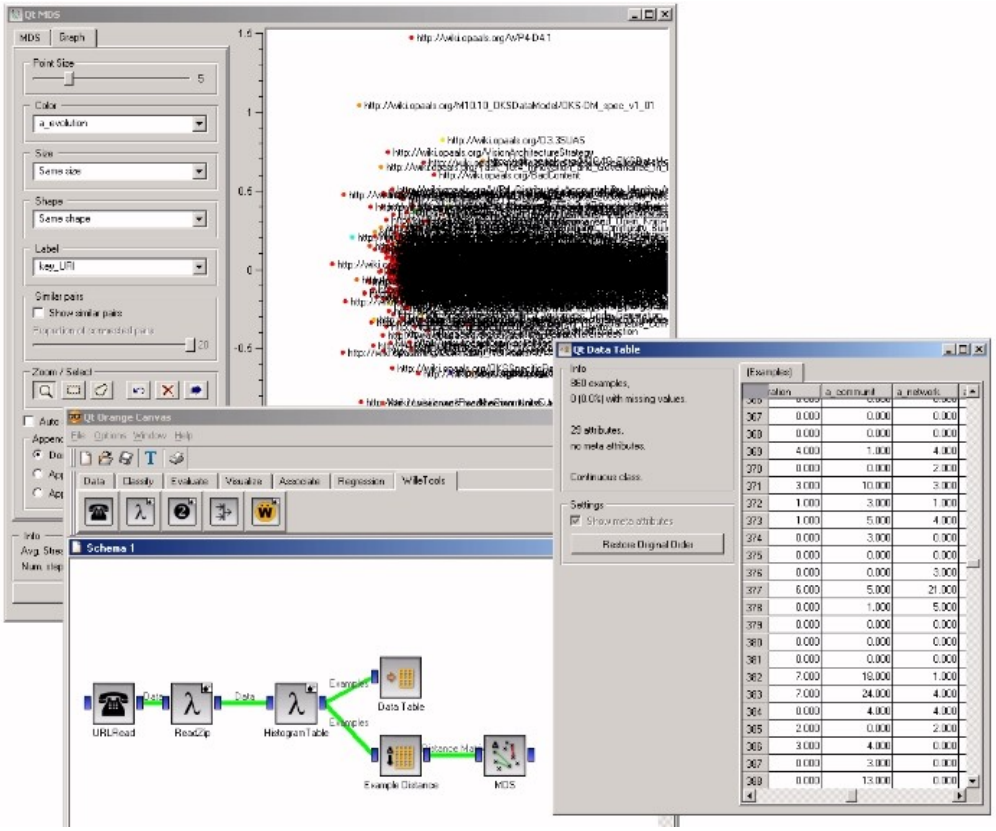


Figure 5.3.3.2. *Interactive Orange widgets for analysing data and visualisations*

Figure 5.3.3.2 illustrates the end-user view to the simple Wille2 Orange application. In practice, the user is able to access full details of the data, and visually navigate within the visualisation interactively. Other kinds of applications may be easily constructed, with the help of built-in Orange data mining and visualisation widgets, and the customised Wille2 widgets.

It is worth observing that while implementing each of the data transformation steps is possible within Orange (or some other data analysis tool), it is not practically feasible. In particular, the process of retrieving data from a large network system may take hours. Thus, using archived data may be required.

Further, even if the data set in the example seems relatively small (=around 1000 instances with some 30 attributes), performing complex iterative algorithms with the data may yield long computations. (Not in this case, though, since MDS is very quick.) This is related to the inherit complexity of data mining applications (=data transformations), not necessarily visualisation applications (=visualisation transformation).

5.3.4. Discussion

From the perspective of abstract modelling, data models can be arbitrary complex. Depending on the data modelling framework, new aspects may be added to the model simply by adding new dimensions (relational modelling) or new types of entities and relationships (graph modelling). In principle, the volume of content data is not a problem either: it merely reflects the size of the table of the graph.

However, from the perspective of visualisations, the attempt of trying to too faithfully illustrate arbitrary complex models leads into visualising things in the terms of another modelling language. In practice, useful visualisations reduce or simplify the modelling language and/or the data by appropriate visual mappings. This means that several complementary (simple) visualisations might be needed.

It is important to note that our visualisation application example might simply be considered as an iterative visualisation application design process, i.e. trying to create a desired visualisation. Alternatively, the process might be considered as a data exploration process, i.e. trying to understand the application data. Of course, both perspectives are nearly always present in practical application development.

However, since the visualisation application is based on the Orange and Wille2 python libraries, the same application can be implemented without the hosting visual programming environment (which could be used for this also, but does not nicely support the use case of publishing services). This allows, e.g., implementing and publishing a similar applications following the HTTP Adapted Component architectural configuration. Thus, effectively, accessible and explorative design-time data modelling and visualisation tools should by default support design and deployment of high-quality end-user applications.

5.4. Distributing and Launching Visualisation Application Executables: Case Visual Web Search

Visual Web Search application is an experimental HTML/AJAX-based visualisation running on Wille2 infrastructure. By using Visual Web Search as an example, we will demonstrate how visualisation application executables can be distributed and launched from various containers.

5.4.1. Visual Web Search

Visual Web Search is an experimental, proof-of-concept HTML/AJAX-based visualisation as an application. The basic idea is to demonstrate how a visualisation can be constructed as a sharable, HTML/AJAX-based web application. Such a way provides easy means for not only sharing visualisations but also to embed applications to third-party applications -- such as OKS Desktop applications in particular.

The application works as follows: when user enters a query, the visual search app retrieves results from Google Web Search, Flickr and YouTube. Instead of textual results, thumbnails from resulting content are generated and displayed to the user. Some of the thumbnails are generated by using components integrated as Wille2 services.

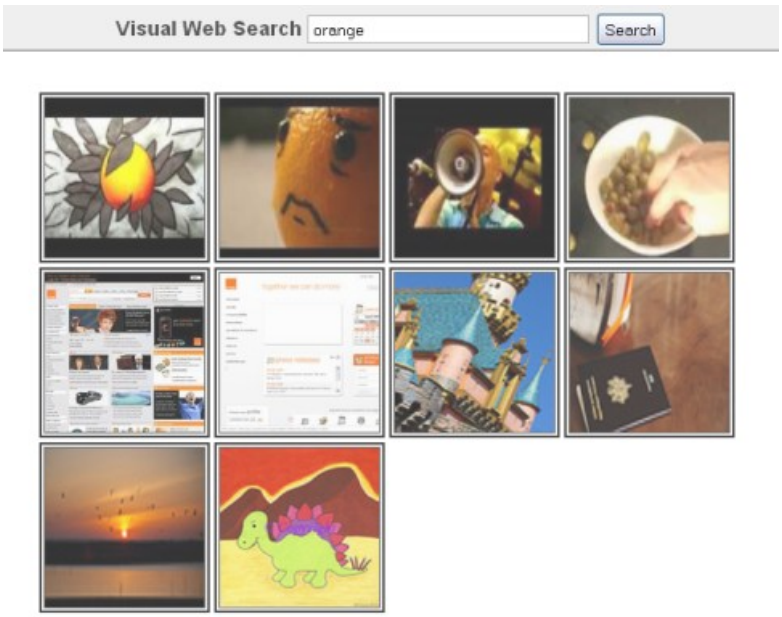


Figure 5.4.1.1. *Screenshot from Visual Search Application, with search term 'orange'*

The basic idea is to implement a visualisation as a standalone application that provides a HTTP server from which the visualisation (user) interface can be requested by the embedding application, as a specialised Wille2 service.

The following fragment of code demonstrates this with an example:

```
import wille.server

urls = (
    ("/media/(.*)", ServeFiles),
    ("/", SearchVis)
)

if __name__ == "__main__":
    server = wille.server.Server(81, services_dir='../../services/')
    server.register_app('search-vis', urls)
    webbrowser.open(server.resolve_app_url('search-vis'))
    server.run(background=False, multi_thread=True, quiet=False)
```

The program logic is straightforward. An instance of Wille2 Server is created, and Wille2 services from specified location are loaded. A new application is registered to the server by invoking *register_app* method. The internal structure of the registered app is defined by given list of URL patterns: each pattern is matched with a callback class that is instantiated to handle requests. A web browser is opened with the app's URL and finally, a web server is launched to serve the given application and services. By default, the server is visible only locally, but with extra parameters, access can be fine-grained to pattern of IP addresses or even set to public.

The actual service is essentially implemented by the two declared handler methods. In this case, SearchVis class implements the visualisation along with Wille2 specific pipeline processing, and writes the HTML/AJAX user interface. The second handler, ServeFiles, is used to create a simple HTTP file server to serve media files required by the user interface.

Of course, not all applications follow the same kind of program logic. For instance, some services my simply provide data objects for external applications while some applications require extensive session management. With this respect, the Wille2 framework aims supporting relatively simple (interface) services. However, more complex applications (e.g. requiring user management and a general-purpose database back-end) may be implemented in a similar fashion with, e.g., the Django framework.

5.5. Designer Collaboration

In this scenario, we will briefly discuss how collaboration between different user groups, as described in section 3.1., could take place by using Wille. We will not focus on contentual side of collaboration, but rather on how different parts of Wille2 can be shared and collaborated on. Based on this view, we discuss collaboration on three levels:

- *Sharing data and services*. Collaboration that is based on shared data or services.
- *Sharing componcents*. Collaboration based on sharing visualisation components, instead of providing them as services.
- *Sharing and embedding visualisations*. Collaboration based on sharing visualisation applications and their design, rather than providing them as services.

5.5.1. Sharing data and services

Technically the most trivial, but in practise a very useful way of using Wille2 for collaboration is to use it to share data or services. We see that this is a very typical form of collaboration: by only sharing static data or access to a service interface rather than sharing the actual components, it is possible to retain more control over ones assets. For example, a social network analysis researcher might want to retain control over full SNA data, while exposing parts of it to trusted partners via well-defined service or data sharing interfaces.

By default, visualisation designer and author collaboration takes place through the regular OKS P2P (Desktop) tools. However, in ad hoc or grass-roots scenarios this might be an overkill. For this reason, the Wille2 library includes implementation of a simple HTTP file server. This allows sharing files without the need of introducing other applications. (Of course, with the cost of some extra installation and configuration work, a similar and more versatile functionality can be achieved by running locally a complete HTTP server, e.g., the Apache HTTP server.)

Listing 5.5.1.1. *Launching and stopping a sample Wille2 fileserver*

```
cd c:\wille2\wille\apps\fileserver\
python fileserver.py 8080 127.0.0.1,130.130.8.9
Starting server to port 8080, URL http://130.230.8.230:8080/
Server visibility: 127.0.0.1,130.230.8.9
Stopping server
```

In the above example, a fileserver is launched in port 8080, providing files from directory `c:\wille2\wille\apps\fileserver\shared`. Access is restricted to the localhost, and requests from 130.230.8.9. The fileserver is shut down when the console window is terminated or Ctrl+C is pressed. When used in parallel with, e.g., a chat tool, the simple fileserver enables rapid prototyping of visualisations, and passing data and components.

Any other services or apps can be shared in a similar fashion: access to Wille2 server can be restricted by a list of IP addresses. While this provides is only with a very rudimentary way of access control, it enables us to use Wille2 to share data and services.

The simple fileserver can be easily integrated in the process visual programming as well. In particular, when working with the Orange Canvas data mining environment, the Wille2 Browser widget allows deploying output from a data and visualisation transformation pipeline into a specific directory, than can then be served to other designers (etc.) using the simple fileserver. Further, data (perhaps published by others) can be read into a Orange Canvas visualisation application, e.g., using the Wille2 URLRead widget. (Of course, achieving a genuine P2P sharing functionality would require appropriately registering data to the OKS (etc.) P2P system.)

5.5.2. Sharing components

With collaboration by sharing components, we refer to the kind of collaboration in which both actual components, e.g. software that is be used to implement services, and component wrappers, e.g. specifications of how components can be wrapped as services, are shared between actors. Motivation for sharing components instead of data or services include: the ability to change service wrapper and specify different ways of using a component as well as the ability to change the component itself (if its shared in source code format).

Let us consider a case in which the actual service description and its implementing component are shared instead of only sharing access to them. As a best-practise, service wrappers are placed under services directory, each on their own subdirectory. Service subdirectory typically also holds all the

components that are required to run the service. For example, *PDF2TXT* conversion service has the following files under `pdf2txt` folder:

- `willeservice.properties` is Wille's service wrapper that describe how components are wrapped and offered as a Wille2 service
- `Multivalent200601.jar` is a JAR (Java Archive) file that contains the actual conversion tool in compiled format

In this case, the component is not available in its source code format and hence can not be modified easily. Also, a less trivial component could consist of larger number of files and folders. We suggest that as a best-practise, components would be deployed as a single archive file, for example in ZIP format.

An archive file can be then easily shared to other actors, e.g. by utilising Wille's built-in fileserver implementation. Another ad-hoc solution would include the use of email for component transfer. When transferred, the receiver can then deploy the component by extracting it under his or her own services folder.

It should be noted that while this approach should work for many components, it is possible that extra steps need to be followed to make some components work. For instance, many command-line service wrappers assume that software external to components has been installed and correctly set up. For instance the above pdf2txt service assumes that Java interpreter can be invoked from working directory. In some systems this may require extra setup or changes to current configuration in order to make the service work.

Another potential issue is in the platform dependence: some components may require operating system dependent tailoring in order to work. For example subdirectories in path variables are separated by a backslash (\) in Windows, but Linux systems usually recognise only (forward) slash (/) as separators. Some of these problems can be tackled by carefully writing a wrapper that is as OS independent as possible. However, without actually testing the components on different platforms, it may be difficult to guarantee their functionality. Also, some components may need to be specifically compiled against the target platform. In this case, an extra step of compiling a component needs to be taken.

In an ideal world, component sharing would be implemented on infrastructural level: a sophisticated P2P OKS platform would allow to easily share components without the need to manually create a ZIP archive or having to reconfigure the component against the underlying OS.

5.5.3. Sharing and Embedding visualisations

An HTTP-based Wille2 user interface service (or the service component) can be run in a stand-alone mode by simply executing its main method. Once up and running, the related visualisations may be requested from the service. The main requirement for the embedding application is that it is capable of running a Web browser window.

Using Web browsers as the default view to visualisations enables implementing many kinds of visualisations and integrating them with other applications, including authoring tools. Further, introducing new concepts and tools for installing and managing viewer extensions and plugins is not required since this can be essentially accomplished with the familiar browser functionality.

Locally (or globally, if required) publishing visualisations as service interfaces provides a nice design pattern for integrating visualisations with several kinds of applications. Further, the executable source code of a visualisation application can simply be distributed as a file folder (see the figure below).

A typical visualisation application includes three components: A folder for application-specific data, the Python executable of the service, and an HTML document, acting as an entrance to the visualisation.

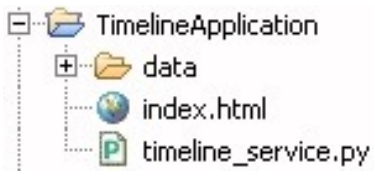


Figure 5.5.3.1. *Visualisation application executable source code in a folder*

In the simplest setting, visualisations can be copied as regular folders and run in the default browser as standalone applications when necessary. In other words, executing visualisations does not necessarily require OKS-specific applications: The resource management tools of the users' operating environment are sufficient. Also, making simple modifications to the visualisation applications is relatively easy.

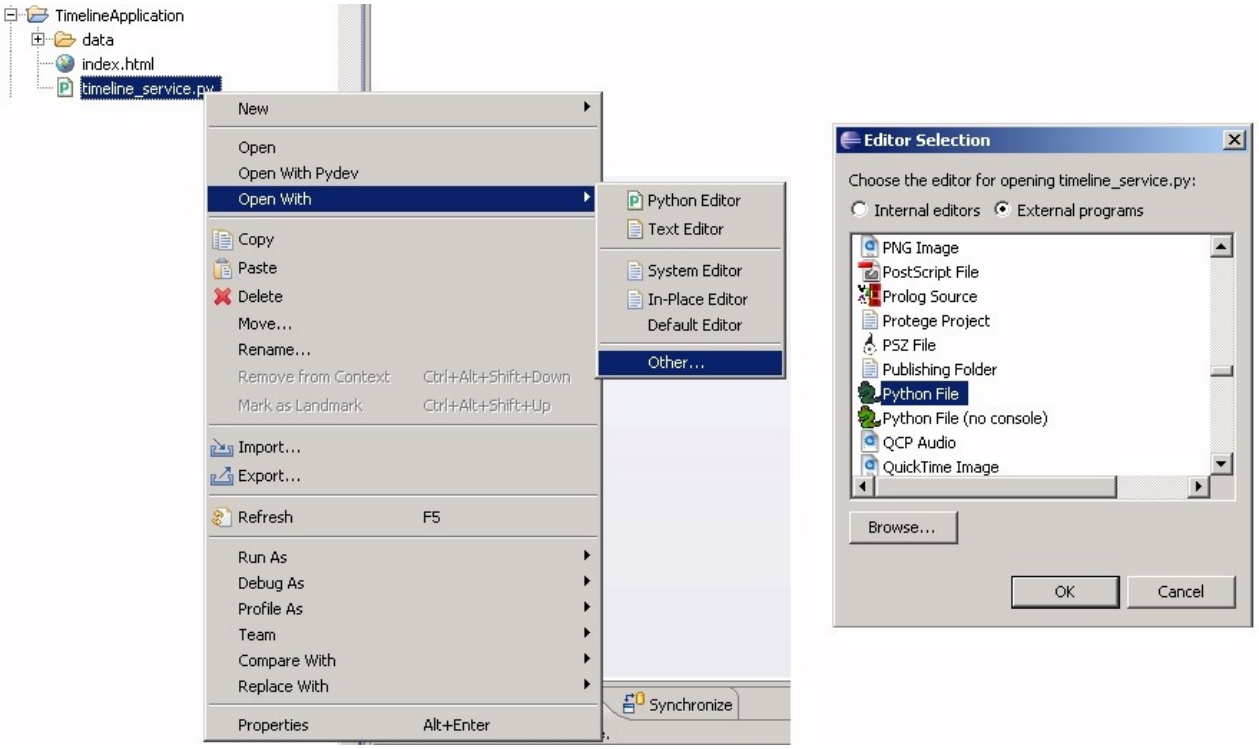


Figure 5.4.2.2. *Launching a visualisation within Eclipse*

In an OKS Desktop application, based on Eclipse Platform, visualisations can be run as services in the local OKS Node. When user interface is required, an appropriate browser view can be requested from the service. This approach enables the integration of Wille2 applications to various kinds of OKS Desktop applications, implemented e.g. with Web technologies or as a dedicated, local desktop application.

The fact that visualisation applications may be realised as Python executables and may be shared simply as folders, provides a generic method for embedding visualisations to other environments as well. For instance, using the Eclipse Navigator or the Package Explorer, visualisations may be opened in Eclipse browser views. Perhaps the simplest way of doing this takes place through selecting the service, launching it externally (see the figure above), and opening the entrance HTML document within Eclipse. (An alternative strategy is registering a new File Association for, e.g., Python executables with the .wille file extension and renaming the Python service executables accordingly.)

If execution of the service within the process control of Eclipse is desired, including, e.g., an appropriate ant task in the visualisation folder, is easy. Note, however, that this may require understanding the process control mechanism of the embedding application. In particular, users must be informed how to shut down the visualisation server is performed since an explicit console window might not be provided. (When visualisations are run externally in a console window, closing the console window shuts down the service.)

A typical configuration of Wille2 on top of Eclipse would require users to separately install a Python interpreter. In order to maintain minimal OS dependence, a Jython interpreter can be used instead of CPython. When using Eclipse, Jython also provides interesting possibilities for more in-depth integration of Wille2 with Eclipse. In theory Jython interpreter could be invoked directly from Java (See http://www.jython.org/docs/embedding.html). Thus, by implementing an additional component for Eclipse in Java, more control over Wille's OKS Desktop integration could be achieved.

5.6. Towards Wille2 Dashboard

The concept of a *widget* is gaining more momentum in the modern Web. Web widgets in desktops (OS X, Windows), TV widgets, mobile widgets (Nokia WidSets, Windows Mobile Widgets, Opera Widgets) all enable users selecting a portfolio of simple views or tools presenting Web-originated information on weather, status updates in social networks and other simple duties. A similar approach is taken in Web *gadgets* that appear e.g. in iGoogle providing a customisable start page or a simple *dashboard* for Google users. Technologies used to develop widgets and gadgets range from traditional Web technology stack (HTML, CSS, Javascript, Flash) to different platform-specific solutions.

In addition to providing a possible delivery platform for visualisations, widgets introduce a workable abstraction to stand-alone visualisation view components. More to the point, composing individual widgets into a *visualisation dashboard* adds to the value of individual views. The dashboard approach follows the principles of Multiple Views interaction technique (Kosara et al. 2003) where the data is visualised with a set of parallel views implementing different visualisation schemes. The examples of possible views include scatterplot, timeline, image collage, graph and others visualisation natives.

5.6.1. Search Dashboard

With Wille2, a tailored dashboard can be developed in a streamlined manner. Figure 5.6.1.1 presents an example of a simple search dashboard composing of three widgets, search box, text-based search results and visual search results.

The Search Dashboard has been developed by integrating two different views representations of data retrieved and represented by Visual Web Search. The left-hand view provides ordinary, text-based rendering of the results. Meanwhile, the right-hand view provides a thumbnail view to the results, and displays details about link such as title, as tooltip texts. The current dashboard would benefit from additional widgets. If information on the publishing date of the resources would be available, for example, a timeline could be added to the dashboard.

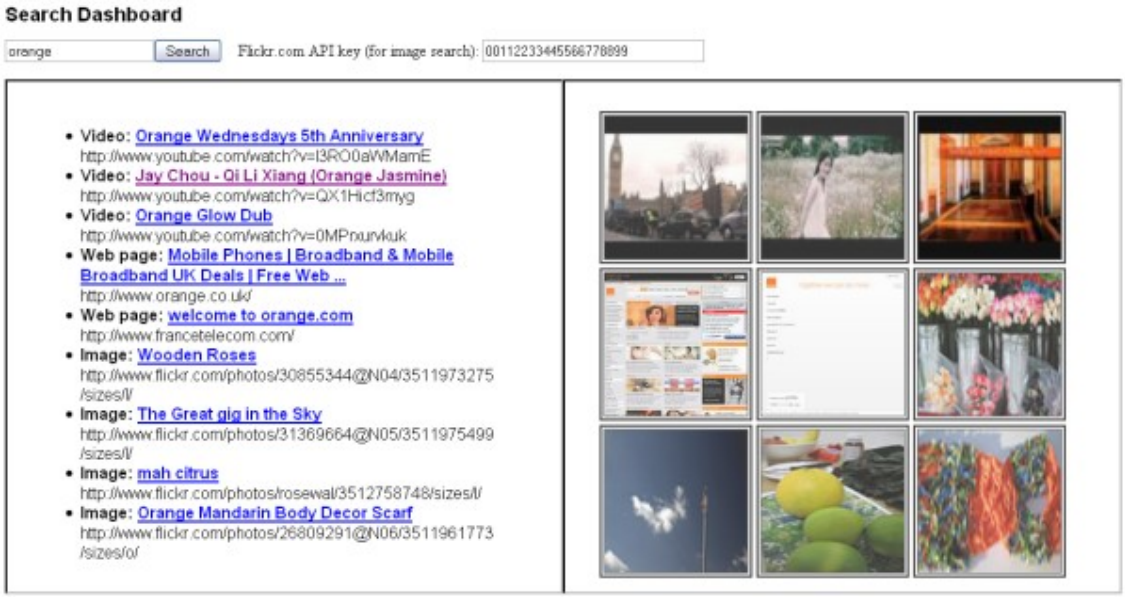


Figure 5.6.1.1. *Example view from the Search Dashboard showing results for 'orange'*

In a conceptual level, Search Dashboard provides a proof-of-concept of the following process:

- A single *service* or a *pipeline* wrapped as a service is used to retrieve data
- Data is passed to applications that render it in different ways

In our implementation, the integration of different views is done utilising *iframe*s: hence, communication between single views or widgets is very restricted. The possible future development paths include a dashboard management functionality enabling end-user driven selection and configuration of individual visualisation widgets, an effort now forced to be done by a visualisation developer. Moreover, in order to enable *Linking and Brushing*, another interaction technique defined by Kosara *et al.* (2003) general means for inter-widget messaging has to be developed. Of course, rather than developing new dashboard technology, it might make sense to adopt an existing one.

The future work related to a visualisation dashboard and widget-oriented visualisation development will continue in OPAALS Phase three.

6. Conclusions

In this deliverable, we have introduced the developed Wille2 visualisation system, explained its main architectural decisions, illustrated its usage, and demonstrated it via a number of scenarios. In short, the Wille2 visualisation system builds upon the first visualisation system prototype, but aims for a more general system. This work establishes the technical basement and provides examples for using the high-level scripting language Python for composing complex data and visualisation processing applications, based on a series of integrated service components.

From developers' and designers' point of view, designing complex data-driven visualisation applications becomes easier. Perhaps the most concrete output of this work is that *with the help of the Wille2 visualisation system, developers can focus on the application-specific data and application-specific user interface, not having to redesign the repetitive tasks of data access, and data and visualisation transformations in every visualisation application*. Also, the scenarios and example applications provide a quick starting point for making complex visualisation applications.

However, it should be obvious that working with visualisations can be surprisingly difficult. Fundamentally, this is related to the complexity of *working with ill-formed data and designing application logic yielding data-driven visualisations*. While visual programming aids (such as spreadsheet or visual pipeline editors) may significantly lower the skill requirements of developers and designers, one cannot really escape programming when automating non-trivial things. These are some of the hard problems of computer science. Escaping this would seem to require either focusing on trivial applications (e.g. fixing authoring tools and visual templates for visualising data), or posing additional (currently unsolved and thus "unrealistic") requirements of delegation and common-sense reasoning to the technical solution (effectively asking computer programs to figure out the necessary programming, e.g., based on examples or narrative descriptions).

We conclude this article with general notes and specific notes about the Wille2 visualisation system, in the context with the presented scenarios. (The reader is encouraged to try out the provided software and view the related technical documentation for gaining more insight related to the observations.)

General notes about the Wille2 Framework:

- *Scripting and visual aids support creativity.* While some visualisation applications can be done solely via "visual programming", implementing the scenario example application would have been very awkward using only a pipeline definition language. This is particularly the case in data pre-processing and itemisation.
- *Visualisations are often developed incrementally.* Rapid prototyping and simple deployment are necessary. Few visualisation applications can be finalised with a monolithic approach. Further, understanding of the data benefits from data exploration.
- *To support interoperability, abstract component interfaces should be favoured.* Since no particular implementation technology suits all needs, gluing different components together should be possible. (Using, e.g. Web APIs.)
- *Typical data and visualisation transformations require several different software components.* When using well-defined component interfaces, e.g., the linguistic analysis of this scenario could be done (better) using tools from WP9. (Wille2 support this by default by allowing the publication software components, wrapped as services.)
- *Visualisation application development requires programming.* It is unrealistic to assume that implementing an arbitrary visualisation application could be done without, e.g., script programming. In complex visualisation applications, also algorithm-intensive development is needed.

- *The viewer component of a visualisation depends upon the application.* The last steps of a visualisation pipeline typically involve mapping data in a format suitable for a particular viewer application (e.g. Orange widget, SVG viewer, X3D viewer, or a Web browser). However, since implementing general-purpose viewers is very resource-consuming, they should be considered as a significant dependency in visualisation design.
- *The schema of the generally used data formats has to be documented in order to ensure it is applied on a correct manner.* Both the overall structure, the occurrence of different parts of the data and document (XML elements and attributes, for example) and the data types used for individual values (e.g. date format) have to be defined. The schema of an XML document, for example, can be defined as a XML Schema or DTD complemented with references to additional data formats. Informal examples are needed to ensure the streamlined usage of the format.
- *Web browser has an important role in Wille*: as web browsers are available virtually on all systems, they can can be easily used as an integrated viewer component. As such, the use of HTML/AJAX user interfaces is very motivated: while more special viewers might be required to achieve some specific visualisation tasks, it is very desirable to make as much of the user interface as possible available in a web browser compliant format. Also modern web browsers enable easy plugin management, especially when it comes to viewer components.

Application-specific notes:

1. OPAALS 2008 workspace scenario
    - *Wille2 can be used as a publishing framework in addition to pure information visualisation.* However, Wille2 is not designed to be a content delivery system.
    - *Ultimately, the dependencies and requirements of a visualisation system are very commonplace.* The creation or collection of visualisation data, for example, should be fostered with a process that is as natural as possible for the actors involved. Wille2 can be used to merge data created and managed with existing tools from traditional spreadsheet applications to online platform. Moreover, as Wille2 provides means to transfer data between systems, it can also be utilised as a light-weight integration technology.
    - *Collecting a set of explicitly represented data is often more resource consuming than the actual visualisation development.* Moreover, often a need to collect additional data emerges along the visualisation development.
2. Social network analysis and visualisation scenario
    - *Different visualisations show the same data in different manner.* The majority of the visualisations, if not all, provide projections to a certain set of data.
    - *With timeline-based animations, an evolving phenomena or dynamic behaviour (intensity, speed) can be better understood.* This resembles to the significance of interaction in supporting the perception of the data (cf. Kosara et al., 2003).
    - *The requirements posed by SNA data analysis differ from the needs of the visualisation-oriented analysis.* A number of tools are useful for SNA work, each supporting different data formats. Thus, different representations of the data must be created. With Wille2, this process can be automated to minimise the work load in data management.
    - *Defining and reusing data formats enables the reuse of individual visualisation components*. Utilising the log file approach, for example, enable the generation of graph representations of social networks emerging in e.g. OPAALS wiki and

OPAALS 2008 community.

- *Some of the steps in a visualisation process may require manual work.* There are existing, useful visualisation or analysis tools that can be integrated only via static data files since they are applications by design. Creating an animation of the evolution of a community, for example, is optimally done by an expert user of SoNIA, Commetrix or some other evolution visualisation tool. After more experience of the process is gained, these manual steps may be sometimes automated.
- *Visualisations may expose delicate information.* SNA visualisation, for example, may brake the boundaries of privacy of individual actors in the visualisation. Either too much data is exposed to the visualisation or the visualisation merely gives insight on certain delicate aspects of the phenomena that are difficult to perceive directly from the data.

3. Data Mining and Visual Programming Scenario
    - *Understanding visualisations may require understanding data transformations.* Many visualisations may be considered as projections. The more complex the projection is, the more end user should understand about the projection. The chief exception is nearly natural sensory representations (e.g. projections onto natural objects or well-known abstractions such as maps) that end users may understand intuitively. (It is a good question what, say, and MDS or SOM visualisation actually means to the end-user.)
    - *Understanding is supported with several complementary visualisations.* Since simple and intuitive visualisations are usually required, end users should benefit from several kinds of visualisations. This also supports validation (both data and understanding). Note that from this point of view, a table view or a Web page are simply kinds of visualisations.
    - *People are better in recognising important structures than explicitly setting criteria for them.* It is often easier to, e.g., recognise a significant property (e.g. regularity or discontinuity) in data via exploratory and iterative means, rather than explicitly trying to define it. A good visualisation also supports creating and testing hypothesis.
    - *Processing large datasets may require archiving data.* Accessing real-time data might be in some cases impossible, e.g., due to network delays or availability.
    - *Processing large datasets may require archived computations.* Computing complex visualisations may take time. Thus, also showing "real-time visualisations" might be in some cases impossible.
4. Visual Web Search Scenario
    - *Wille2 can be used as a lightweight solution for publishing services/apps over HTTP*. However, it does not substitute nor remove the need for production-level solutions to HTTP or P2P service publication. For more stable and scalable web/P2P publishing solutions, use of production-ready systems such as P2P service infrastructure or web services, could be motivated.
    - *Publishing services over HTTP enables us to create reactive visualisation applications*. As such, a visualisation is not required to retrieve all data prior to its use, but can, instead, request data on-demand. This also enables also to lower the latency of visualisation applications: view rendering is not anymore dependent on the amount of data processed, but instead depends on the latency of the network.
    - *As existing Web services provide access to data and services that are potentially*

useful, the ability to integrate to them is important also in Wille; they enable us to build application that are useful right here, right now. In later stages, it is desirable to convert at least some of the services to run on top of P2P infrastructure, but it should not be assumed that "legacy" web APIs would cease to exist after P2P's adoption. In fact, switching to decentralised P2P solution may be even against e.g. business interests of a service or data provider.

- *Legal and related copyright and access issues seem to be a common concern*, when adapting existing services to Wille2 or P2P infrastructure. For example Google Search API's terms of use are very restrictive, making it almost impossible to build visualisations on top of its data[1]. On the other hand, the use of screen-scraped data is usually restricted by copyright. In general, it seems that the use of web APIs is very commonly limited to personal or non-commercial use, or - maybe even worse - is not specified at all, leaving legal considerations up to their users.

5. Collaboration Scenario
    - *The ability to share files and services is seen very important for ad hoc or grass-roots collaboration*. Other approaches, such as using a full-blown publishing framework or web server, would be definitely too heavy on scenarios where focus is on rapid and instant sharing of data.
    - *Best practise of sharing components as single archive files is feasible for many cases, but does not come without complications*. Potential problems include issues with OS dependence or hidden assumptions that wrappers make about their runtime environments.
    - *Support for automated testing of different components and their integration might provide a value-add to Wille*. At the moment, components and services testing on all levels relies on manual work. In the future, it might be desirable to add automated testing support for improved reliability.
    - *Web browser has a very important role in collaboration*. When services and applications are released on top of HTTP, web browser can be used to access them either from a local or a remote machine, access settings abiding. Lightweight integration to OKS Desktop tools, such as Sironta, can be also achieved by using web browser as a service/visualisation container.

As a conclusion, the ability to encapsulate, access and compose general-purpose visualisation components (perhaps as services) is fundamental both in developer collaboration and in joint development of complex visualisation applications. Interesting visualisation applications can be conceptualised as configurations of some basic data accessing and visualisation processing components, glued together with relatively short scripts. Thus, as the pool of available services increases, so increases the potential of creating and accessing more useful visualisation applications with reasonable efforts. However, in order to be truly successful, the community developing visualisations should aim for convergence in numerous related activities: Published data formats, license rules, application processes, etc.

1 See http://code.google.com/intl/fi/apis/ajaxsearch/terms.html

Finally, some (obvious) potential areas of future work related to Wille2 Visualisation System include:

1. **Application development.** E.g. developing case-specific visualisation applications with Wille2.
2. **Extending existing functionality.** E.g. implementing new service application programming interfaces to the Wille2 core.
3. **Usage as a technology platform.** E.g. using Wille2 for implementing general-purpose data processing tools (such as XProc processors).
4. **Best practice development.** E.g. developing or migrating into a general-purpose dashboard and widgets for Wille2 GUI design (for the HTTP Adapted Component architectural configuration).
5. **Adding new functionality.** E.g. adding orchestration-level functions to the Wille2 core or built-in support for certain Wille2 (Web) widgets.

It is important to note that in many cases, adding "new functionality" (items 4 and 5 above) in fact takes naturally place on the application level, as best practices (see Figure 4.1.1 and the block "Other App-Spefic Modules"). For instance, many visualisation applications need to create bitmap and vector graphics from data, using abstract declarative syntax. This can be technically achieved in applications simply by importing appropriate Python modules to the application (e.g. SVG). This design philosophy applies to a number of Wille2 use cases and required functionality, aiming to keep the Wille2 core elegant and understandable.

Indeed, some of the above topics are included as work topics in the third phase of the instrumental OPAALS project.

Acknowledgements

This work is a result of collaborative work of the OPAALS project team at TUT. In particular, we appreciate the collaboration of Thumas Miilumäki, and the past contributions of Matti Haapaniemi, Antti Kortemaa, Arto Liukkonen, and Markus Mannio. Finally, we welcome and value the feedback and the formal and informal input from the other contributing project partners.

References

Card, S. K., Mackinlay, J., & Shneiderman, B. (1999). Readings in Information Visualization: Using Vision to Think (1st ed., p. 712). San Francisco, CA, USA: Morgan Kaufmann.

Desodt, T. (2007). Deliverable 10.4: A demonstration of the OKS Technical Platform. OPAALS (IST-034824).

Geroimenko, V. & Chen, C. (Eds) (2005). Visualizing Information Using SVG and X3D. Springer, UK.

Grimnes, G. (2006). Semantic Conference Program. In: 3rd European Semantic Web Conference. Budva, Montenegro (2006), http://www.eswc2006.org/technologies/designchallenge/semantic-conference-program.pdf

Haapaniemi, M., Huhtamäki, J., Kortemaa, A., Mannio, M., Nykänen, O. & Salonen, J. (Eds.). (2007). Deliverable D10.6: A proof-of-concept implementation of a visualisation client. OPAALS project deliverable, Phase 1.

Hearst, M. A., & Rosner, D. (2008). Tag Clouds: Data Analysis Tool or Social Signaller? In Proceedings of the Proceedings of the 41st Annual Hawaii International Conference on System Sciences (p. 160). IEEE Computer Society. Retrieved October 31, 2008, from http://portal.acm.org/citation.cfm?id=1334989.

Huhtamäki, J. (2007). Community visualisations in Open Knowledge Space: Uncovering rabbit holes in a digital ecosystem. In Proceedings of 1st OPAALS workshop, Noveber 26-27, 2007, Rome, Italy. http://opaals.org/.

Huhtamäki, J., Nykänen, O., Salonen, J. (Eds.). (2009a). Deliverable D10.11: Specification of the P2P configuration of the visualisation system. OPAALS project deliverable, Phase 2.

Huhtamäki, J., Nykänen, O., & Salonen, J. (2009b). Catalysing the Development of a Conference Workspace. In HCI International 2009 Conference Proceedings, San Diego, CA, USA, July 19-14, 2009. Berlin/Heidelberg: Springer. To appear.

Keim, D. (2002). Information visualization and visual data mining. Transactions on Visualization and Computer Graphics, 7(1), 100-107.

Kosara, R., Hauser, H., & Gresh, D. (2003). An Interaction View on Information Visualization. In State-of-the-Art Proceedings of EUROGRAPHICS 2003 (pp. 123–137).

Kurz, T., Heistracher, T. J., Eder, R. (2008). Deliverable 10.7: Visualisation Service for P2P infrastructure and EvESim based on GoogleMaps. OPAALS (IST-034824).

Lapteva, O., Peukert, H., Nykänen, O. & Eder, R. (2009). Deliverable 6.8: Models of the Evolutionary Framework of Language. OPAALS (IST-034824).

Möller, K., Heath, T., Handschuh, S., & Domingue, J. (2008). Recipes for Semantic Web Dog Food — The ESWC and ISWC Metadata Projects. In The Semantic Web, Lecture Notes in Computer Science (Vol. 4825, pp. 802-815). Berlin/Heidelberg: Springer. Retrieved June 30, 2008, from http://dx.doi.org/10.1007/978-3-540-76298-0_58.

Nykänen, O. (2007a). Interpretation Logics. Proceedings of the 1st OPAALS conference, 26-27 November 2007, Rome, Italy. Available: http://matriisi.ee.tut.fi/hypermedia/julkaisut/2007-nykanen-ilogics.pdf

Nykänen, O. (2007b). Implementing Context-Specific Views to Distributed (Rule) Databases with Fuzzy Logic, Proceedings of the International IADIS, WWW/Internet 2007 Conference, 5-8 October, Vila Real, Portugal, pp. 305-312.

Nykänen, O. (2009). Semantic Web for Evolutionary Peer-to-Peer Knowledge Space. In Birkenbihl, K., Quesada-Ruiz, E., & Priesca-Balbin, P. (Eds.) Monograph: Universal, Ubiquitous and Intelligent Web, UPGRADE, The European Journal for the Informatics Professional, Vol. X, Issue No. 1, February 2009, ISSN 1684-5285, CEPIS & Novática. Available at http://www.upgrade-cepis.org/issues/2009/1/upgrade-vol-X-1.html

Nykänen, O., Huhtamäki, J., Salonen, J., Pohjolainen, S., & Silius, K. (Eds.). (2008). Proceedings of the 2nd International OPAALS Conference on Digital Ecosystems: OPAALS 2008. 7-8 October 2008, Tampere, Finland. 108 pages. Available: http://matriisi.ee.tut.fi/hypermedia/julkaisut/opaals2008-proceedings.pdf

Nykänen, O., Salonen, J., Haapaniemi, M., & Huhtamäki, J. (2008a). A Visualisation System for a Peer-to-Peer Information Space. Proceedings of OPAALS 2008, 7-8 October 2008, Tampere, Finland. pp. 76-86.

Nykänen, O., Salonen, J., Huhtamäki, J., & Haapaniemi, M. (2008b). OKS Data Model, version 1.01. A milestone specification for the OPAALS PROJECT (Contract number IST-034824), WP10: Sustainable Research Community Building in the Open Knowledge Space. Contribution to the Milestone M10.10: OKS Data Model (M24), 1 July 2008 (29 pages). Available: http://matriisi.ee.tut.fi/hypermedia/julkaisut/20080701-oks-dm-v1-01.pdf

Nykänen, O., Mannio, M., Huhtamäki, J. & Salonen, J. (2007). A Socio-Technical Framework for Visualising an Open Knowledge Space, Proceedings of the International IADIS WWW/Internet 2007 Conference, 5-8 October, Vila Real, Portugal, pp. 137-144.

Smith, K., Seligman, L., Swarup, V. (2008). Everybody Share: The Challenge of Data-Sharing Systems. IEEE Computer Magazine, September 2008.

Steinmetz, R. & Wehrle, K. (2006). What is This "Peer-to-Peer" About? In Steinmetz, R., & Wehrle,K. (Eds): Peer-to-Peer Systems and Applications. Lecture Notes on Computer Science, Springer.

Telea, A.C. (2008). Data Visualization. A K Peters Ltd., USA.

Ware, C. (2004). Information Visualization: Perception for Design (2nd ed., p. 560). San Francisco, CA, USA: Elsevier.

Witten, I. H., Frank, E. Data Mining: Practical Machine Learning Tools and Techniques. Second Edition. (2005). Morgan Kaufmann Publishers, Elsevier Inc.

Appendix A: Developer's Documentation and Catalog of Wille2 Orange Widgets

Wille Orange widgets integrate the basic functionality of the Wille visualisation development framework to the component-based data mining software Orange. Once installed, the new widgets are grouped in the WilleTools tab in the Orange Canvas application (http://www.ailab.si/orange/).

This document introduces the new Wille2 widgets and includes example codes for script/schema developers. It is assumed that user of Wille Orange Canvas widgets (and the reader) has the basic knowledge about the Orange and the Orange Canvas visual programming application.

1. Introduction

In short, the WilleTools widgets enable adding custom program logic, service requests, rich content (e.g. XML) management, and third-party applications to Orange applications. The motivation is using Orange Canvas as an exploratory, visual programming environment in developing relatively general-purpose data processing, semantic computing, and visualisation applications. Note that since Orange Canvas is not designed as a service platform (and does not scale as such), it is expected that mature applications are designed so that they can be re-factored in terms of the underlying Orange and Wille libraries, to be deployed (perhaps as service components) without the visual Canvas application.

The design of Wille Orange widgets has followed the idea of scripting. Further, it is assumed that developers of scripts know what they are doing. As a consequence, Wille widgets in principle support processing and transmitting arbitrary Python `objects`. While this allows implementing nearly anything within the Canvas application, the drawback is that the signal type checking paradigm of Orange Canvas has been weakened within the WilleTools widgets. This means that Script widgets can be connected in the Canvas even if they do not support each others signal types. In most cases, however, Wille widgets behave as expected.

For purpose of working with folder-like objects, typically required in complex pipeline applications, many of the Wille2 widgets are based on an utility class called `wille.zipblob.Zipblob`, as an application of the common Python Zipfile object. This allows manipulating and transmitting collections of data as compressed ZIP files. For compatibility reasons, using Unicode file names is not recommended. Further, due to the limitations of different ZIP versions, a Zipfile can only include data up to 2GB. However, it should be possible to pack/unpack Zipblobs using common ZIP tools.

Important security note: It is expected that developers share the Orange Canvas schemas (and perhaps new widgets) they have created. In this setting, introducing widgets capable of executing arbitrary Python (and therefore other) programs raises an obvious security risk. For this reason, the Wille Orange widgets with potentially harmful computing capabilities are clearly marked with a little bomb icon  (a small bomb in a white frame on top right corner of the widget icon). However, this is *merely a remainder* that downloading 3rd party schemas including related widget code may potentially include malware. Thus, *when using Wille Orange Canvas widget extensions, never run untrusted canvas schemas!* Obviously, the developers of the Wille Orange widgets take no responsibility whatsoever from damage resulting from running malicious code from an untrusted source. (Of course, widgets may also contain other errors/caveats that open additional, undocumented security threats.)

2. Installation

Installation procedure of Wille2 for Orange Canvas is straightforward (details may vary slightly upon operating system, installation instructions written for Wille 2.0 Bundle):

1. If necessary (e.g. not already installed), install **Python** 2.5(+) (see http://www.python.org/download/). An alternative is to use, e.g., ActivePython (see http://www.activestate.com/activepython/).
2. If necessary, install the latest copy of the **Orange** Canvas (see http://www.ailab.si/orange/downloads.asp) and the Python libraries it requires (Python version 2.5 was used during Wille development). In many cases, this can be achieved in a single step with an appropriate Orange bundle.
3. Download a copy of the Wille distribution archive (see http://www.tut.fi/hypermedia/en/publications/software/wille/). It includes three directories, `wille`, `WilleTools` and `icons`. (The two latter directories can be found in the `apps/orange` directory.) The first is the general-purpose Wille2 Python library, the second is the Wille2 Orange widget library, and the third includes the icons of the Wille2 widgets.
    1. If necessary, copy the directory `wille` (i.e. including the directory itself, not just the contents) into the `site-packages` directory of your Python installation. (In many cases, the destination folder looks something like `...\Python\Lib\site-packages` .)
    2. Copy the directory `WilleTools` (i.e. including the directory itself, not just the contents) into the `OrangeWidgets` directory of your Python Orange Canvas installation. (In many cases, the destination folder looks something like `...\Python\Lib\site-packages\orange\OrangeWidgets` .)
    3. Copy the contents of the directory `icons` (i.e. just the files, not creating the directory) into the `icons` directory of your Python Orange Canvas installation. (In many cases, the destination folder looks something like `...\Python\Lib\site-packages\orange\OrangeWidgets\icons` .)
4. Install the Python libraries required by some of the Wille2 widgets: **4Suite** XML tools (see http://4suite.org/?xslt=downloads.xslt or http://sourceforge.net/projects/foursuite/). **Note:** In some platforms (including Windows), using 4Suite requires running Python (and thus Orange Canvas) in a case-sensitive imports mode (in Windows, this may done by setting `set PYTHONCASEOK=`).
5. Install any additional (Python or other) libraries and components required by your scripts.
6. Finally, registering the new WilleTools widgets is required. This is done within the Orange Canvas application: Launch the Orange Canvas application and choose `Options/Rebuild Widget Registry` from the menu. The WilleTools tab appears now in the Canvas application.

The above procedure needs to be done once per installation. If you have several python systems installed, be careful in avoiding mismatching installations.

Once the everything has been successfully installed, the Python Orange Canvas application is run as any Python GUI application. (Note, however, that using 4Suite may require setting the PYTHONCASEOK environment variable. Since many python applications do not require this, it may be convenient to launch the Canvas application via a shell script, setting the variable only for the session.)

We shall next introduce the new Wille2 widgets, now accessible from the Orange WilleTools tab.

3. Wille2 Widgets

The Wille2 Widgets include the Script Widget, the URLRead Widget, the ZipblobMerge Widget, the XSLT Widget, and the Browser Widget. We shall next introduce these from application developers' perspective.

3.1. Script Widget

The Wille Script widget allows adding arbitrary Python scripts to Orange applications. **Additionally required Python libraries:** By default none, i.e. depends on the script.

Script can both input and output arbitrary data as an Python *object*. In addition, Scripts may output native Orange types, including *ExampleTable*, *AttributeList*, and *SymMatrix*. This allows using Scripts for not only ad hoc computing, but also for gluing different things together. Note that Scripts are the Swiss Army Knives of Wille Orange: When a script becomes generally useful, it is usually not too difficult to use it as a basis for implementing a new kind of widget. (For writing Python scripts, see http://www.python.org/doc/ .)

Controls of the Script dialog include:

- Controls displaying the script name, status, and error information.
- A field for using external script from a file, instead of the provided script edit box. (Usually it makes sense to include Scripts in the edit box since this includes the script when performing the Canvas File/Save operation.)
- A checkbox for stating that a script is runnable.
- A checkbox for setting the script to be run automatically, when new input is signaled.
- A button to execute the script manually (using the last stored input).

A Script receives its input Data signal via the global input variable. It signals output data via the global output dictionary using output signal types as keys (i.e. Data, Examples, Attributes, or Distances). Note that in order to simplify script development, Scripts accept only a single input. When multiple input signals are required, using ZipblobMerge widgets is recommended.

The following simple Script example accepts a Zipblob object that contains an appropriate XML table data file, and outputs the data from the file as an Orange ExampleTable (and can hence be connected with other built-in Orange widgets). The comments are not necessary but may help other users in using and modifying the script.

```
######################################################################
# XMLTable to Orange ExampleTable Converter
#
# Input:  wille.Zipblod.read()able data wrapped using the
#         wille.zipblob.Wrapper class (e.g. )
# Output: Orange ExampleTable
#
# Arguments: INFILE: Name of the source XML file in the input Zipblob

INFILE = "table.xml"

#
# Input example:
#<table>
#  <tr>
#    <th class="StringVariable">Item</th>
#    <th class="FloatVariable">Weight</th>
#  </tr>
#  <tr>
#    <td>Duck</td> <td>3.5</td>
```

```
#  </tr>
#  <tr>
#    <td>Witch</td> <td>3</td>
#  </tr>
#</table>
#
# The format resembles simple HTML tables. However, it
# is assumed that the first row introduces the variables,
# the other rows include the instances, and that
# each row includes the same number of cells.
#
# Script by Ossi, 2009
############################################################################


############################################################################
# Do not make modifications below unless you want to edit the script.

__title__ = 'XMLTable to ExampleTable Converter'

import orange
from wille import zipblob
import xml.dom.minidom

global input, output # Every Wille2 IO script includes these two lines:
output = dict()      # Important! Forgetting this gives a hard-to-track type error!

b = zipblob.Zipblob()
b.write(input.get_value()) # Input Zipblob data from the global input variable

dir = b.check_out()
doc = xml.dom.minidom.parse(str(dir) +'/' + INFILE)
b.check_in()

root = doc.documentElement
trlist = root.getElementsByTagName("tr")

# Create orange table
vars = []
vartypes = []

if len(trlist)==0:
  output["Examples"] = None
else:
  first = 1
  for tr in trlist:
    if first == 1:
      first = 0
      thlist = tr.getElementsByTagName("th")
      for th in thlist:
        type = th.getAttribute("class")
        name = th.firstChild.nodeValue
        if type == "FloatVariable":
          vars.append(orange.FloatVariable(str(name)))
          vartypes.append("float")
          print float
        if type == "StringVariable":
          vars.append(orange.StringVariable(str(name)))
          vartypes.append("string")
      domain = orange.Domain(vars)
      data = orange.ExampleTable(domain)
    else:
      tdlist = tr.getElementsByTagName("td")
      instance = []
      ind = 0
      for td in tdlist:
        value = td.firstChild.nodeValue
        if vartypes[ind] == "float":
```

```
          instance.append(float(value))
        if vartypes[ind] == "string":
          instance.append(str(value))
        ind = ind + 1
      ex = orange.Example(domain, instance)
      data.append(ex)

output["Examples"] = data
# Every Wille2 script includes this line.
# Now output data using the Examples signal # type as ExampleTable.
```

The next simple example Script creates a Zipblob object and signals it forward:

```
from wille import zipblob

__title__ = "Create Zipblob example"

global input, output
output = dict()

b = zipblob.Zipblob()
b.add_file('somefile.txt', 'file.txt', 'a') # add file somefile.txt as file.txt

output["Data"] = b.wrapper_read() # Important: Pass Zipblob data via Wrapper class,
not as such (for garbage collection!)
```

For simplicity, the editor/debugging capabilities of Scripts are very limited. It might thus be useful to develop and test complex scripts using appropriate external development framework, and simply copy/paste them into Wille Orange Scripts when ready.

Developer note: Since Orange Canvas seems to exploit signals for internal bookkeeping, e.g., passing flat string data is may in general be not safe and may yield strange error messages. Arbitrary data can be wrapped into, e.g., classes that seem to work fine.

- TThe wille.zipblob.Wrapper class is provided for this purpose. This allows doing:

```
output["Data"] = zipblob.Wrapper("some text") ... input.get_value().)
```

However, the drawback is that in some application/platform combinations, Python garbage collection does not send `__del__` to classes passed using signals in Orange Canvas. This means that classes requesting system resources might not be able to release resources as expected. In particular, if signalled data is not properly wrapped, this may yield into both resource leaking and a security risks in a worst-case scenario. Of course, this may also happen when a program exists unexpectedly (e.g. crashes). For instance, if signalled as such a wille.zipblob.Zipblob object may leave processed zip files (e.g. *1234567_zipblob.zip*) in the user's temporary system folders. For this reason, the proper way of passing Zipblobs is via wrappings. In other words, instead of passing a Zipblob object directly, it's data is passed via a Wrapper object provided by `Zipblob.wrapper_read()`. The drawback in this is that instead of passing files, the data gets signalled via the computer (virtual) memory.

Security warning: Scripts can implement genuine Python computer programs. Thus, Scripts of untrusted origin may include malware!

3.2. URLRead Widget

The Wille URLRead widget allows reading data from a HTTP server. **Additionally required Python libraries:** `wille`

Controls of the URLRead dialog include:

- Controls displaying the URLRead status and invoke/error information.
- Three fields for inputting the URL address of the service, and username/password, if required.
- A checkbox for setting the URLRead to be executed automatically, when new input is signaled.
- A button to invoke the URLRead request manually (using the last stored input).
- A radio button for HTTP request method type (GET or POST).

URLRead receives its input via input Data signal, typically prepared by a Script. This data should be a `dict` of parameters wrapped using the `wille.zipblob.Wrapper` class. The following Script example demonstrates creating parameters:

```
from wille import zipblob
import urllib

global input, output
output = dict()

params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
output["Data"] = zipblob.Wrapper(params)
```

If present and properly formed, the input data will be passed as parameters to the service.

URLRead outputs the data it receives from the service using the `Data` signal. The data is packed as a file inside a `zipblob.Zipblob` class whose data is then wrapped using the `wille.zipblob.Wrapper` class. Actual data is stored in a single file, called `results`.

If the service returns a single file, the file results may be useful as such. However, depending on the requested service, the output data may require additional parsing (e.g. when multipart or zipped documents were returned).

In any case, the result from the service includes the data returned by the standard read() method requested from an object instance (derived from a class of type) of type `urllib.FancyURLopener`. In other words, the results file includes data that conceptually includes the bytes from the `output_from_service` variable of the following informative example:

```
import urllib
opener = urllib.FancyURLopener({})
f = opener.open("http://www.python.org/")
output_from_service = f.read()
```

It is the task of the next widget, typically a Script widget, to process the output data from URLRead. However, several other Wille Orange widgets by default work with wrapped Zipfile objects.

Developer note: Wille2 library includes more classes and method for invoking and publishing services. This functionality may be exploited using Wille Scripts. Future Wille version may include Orange widget representations of these.

3.3. ZipblobMerge Widget

The Wille URLRead widget allows listing the contents of a Zipblob file, and merging Zipblobs. **Additionally required Python libraries:** `wille`

Controls of the ZipblobMerge dialog include:

- Controls displaying the status and information.
- Contents listing, showing the files and file sizes.

Note that all controls are informative only -- the widget is completely operated via the Orange Canvas interface, by connecting and disconnecting signals.

URLRead receives its input via input wrapped Zipblob Data signal(s), and returns output(s) via wrapped Zipblob data signals. If inputs Zipblobs include files with identical names, the files will be repeated in the merged Zipblob, due to its Python Zipfile functionality.

3.4. XSLT Widget

The Wille XSLT widget allows transforming XML data with Extensible Stylesheet Language Transformations (version 1.0; see http://www.w3.org/TR/xslt). This allows, e.g., data transformations, visualisation transformations, and user interface transformations. **Additionally required Python libraries:** wille, Ft.Xml (from the 4Suite XML Tools).

Controls of the XSLT dialog include:

- Controls displaying the XSLT status and error information.
- Two fields for inputting the relative name of the source file, and the relative name of the output file of the transform (within the Zipblob)
- A checkbox for setting the URLRead to be executed automatically, when new input is signaled.
- A field for using external transformation from a file, instead of the provided edit box.
- A checkbox for stating that XSLT code is runnable.
- A checkbox for setting the transformation to be executed automatically, when new input is signaled.
- A button to execute the transformation manually (using the last stored input).

The following simple transformation example demonstrates creating a HTML 4 document based on the given XML source (with root element called `data`, including `item` elements with textual content).

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
  <xsl:output method="html"
              indent="yes"
              encoding="iso-8859-1"
              doctype-public="-//W3C//DTD HTML 4.0 Transitional//EN" />

  <xsl:template match="/data">
    <html lang="en">
      <head>
```

```
        <title>Example</title>
      </head>
      <body>
        <h1>List of items</h1>
        <ol>
          <xsl:apply-templates/>
        </ol>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="item">
    <li>
      <xsl:value-of select="."/>
    </li>
  </xsl:template>
</xsl:stylesheet>
```

Security warning: In principle, XSLT 1 transformations may read any files the user has access to (and call XSLT-processor-specif extension functions). Further, the XSLT widget supports writing relative filenames. However, since global filenames are useful in some applications, using them is *not* disabled.

3.5. Browser Widget

The Wille Browser widget allows opening a file from a Zipblob in the default browser. In some operating systems it is also possible to open the Zipblob as a file folder.

Additionally required Python libraries: wille.

By default, the folder is placed in the (system-depended) temp file directory of the user. The name of this folder changes with different pipeline executions. To enable viewing the contents of the Zipfile, the temporary directory is maintained until the Browser widget dialog is closed manually. **Note:** If the dialog is not closed manually, the Zipfile directory is exists also after exiting the Orange Canvas application!

It is also possible to force the data into a fixed directory, specified in the Browser Widget dialog. In this case, the same directory will be overwritten as new data signals are available. (This is particularly useful e.g., in refreshing and publishing content to external applications such as in visualisation interface testing or file service deployment.)

Controls of the Browser dialog include:

- Text field for information.
- A checkbox for using a fixed base output directory. (For safety reasons, data will be (over)written to a subfolder called `output`.)
- A button to activate a find directory dialog.
- A button to manually deploy received data in the fixed output directory.
- A checkbox to for deploying data automatically.
- Input text field for specifying the file inside the Zipblob. (If empty, opening the folder is attempted.)
- A checkbox for using the fixed directory instead of the temporal one.
- A checkbox to for launching the default browser automatically.
- A button to launch the default browser.

Security warning: When used in a complex pipeline, it is in principle possible to use the Browser widget for opening harmful applications and/or sending local user data to outsiders. Further, by specifying a fixed output directory, it is possible to overwrite use files in some output directory.

Developer tip: When used with a fileserver application, the fixed directory option provides a simple way to distribute outputs of pipelines within a developer group etc.

4. Discussion

The Wille Orange Widget library aims including the minimal set of useful tools for creating data transformations and visualisations. In particular, several kinds of applications can be implemented using the Script widget.

It is anticipated that the widgets are also useful for building, e.g., Semantic Web applications. Again, with the help of the Script Widget, it is in principle easy to integrate with, say, Resource Description Framework (RDF) data.

For instance, assuming that the Redland (see http://librdf.org/) RDF library with Python bindings is installed, making semantic queries is simple. The following piece of code demonstrates creating an Orange ExampleTable from the results of a SPARQL (select) query (http://www.w3.org/TR/rdf-sparql-query/), as implemented by the Redland query processor:

```
import RDF
import orange

model = RDF.Model()
model.load('file:data.xml'); # Perhaps read from multiple Zipfiles, due several
                             # URLReads, integrated with ZipblobMerge...
qtxt = "select $uri $title where { $a a <http://purl.org/rss/1.0/item>. $uri
<http://purl.org/rss/1.0/title> $title} limit 5"
q = RDF.Query(qtxt, query_language="sparql")

results = q.execute(model) # Should really be prepared for catching exceptions ...

data = None
vars = []
i = 0  # Tuplet id (not used below)
if len(results)>0:
    # make orange string variables
    for result in results:
        for k in result.keys():
            if k.startswith("f_"): # make continuous (float) variable
                vars.append(orange.FloatVariable(k))
            else:
                vars.append(orange.StringVariable(k))
        break
    domain = orange.Domain(vars)        # make domain based on the above variables
    data = orange.ExampleTable(domain)  # make data object
    # compile and append individual data examples
    for result in results:
        i = i + 1
        instance = []
        for k in result.keys():
            if k.startswith("f_"): # make continuous (float) variable
                try: # no value if conversion fails!
                    f = float(str(result[k]))
                except Exception, (errValue):
                    f = ""
                instance.append(f)
            else:
                instance.append(str(result[k]))
```

```
        ex = orange.Example(domain, instance)
        data.append(ex)

# Got i tuples now held in an ExampleTable.
# Could be passed, e.g., to some Orange a widget or method
```

Besides data (semantic) data transformations, the Wille2 Orange widget library support visualisation transformations. Perhaps the simplest way of creating user interfaces is generating graphical end user interfaces with Scripts or XSL Transformations in HTML/AJAX/X3D/SVG/... technologies within the default Browser. More complex applications typically benefit from adopting some Dashboard programming schema.

This also simplifies the socio-technical challenge of persuading users to install required additional plugins etc. In many cases, suitable Browser extensions may already be available, or at least installation and add-on management is relatively simple, following the familiar Browser plugin management use case.

Appendix B: Complete Example Listings of OPAALS 2008 data

Listing 5.1.1.1 *Example of OPAALS member data in FOAF (full version)*

```
<foaf:Person rdf:about="mailto:huhtis@opaals.org">
      <foaf:img rdf:resource="http://wiki.opaals.org/huhtis/FactsAboutPerson?
ction=AttachFile&amp;do=get&amp;target=opaals_fact_person_image.jpg"/>
      <foaf:name>Jukka Huhtamäki</foaf:name>
      <foaf:mbox rdf:resource="mailto:jukka.huhtamaki@tut.fi"/>
      <foaf:Organization rdf:resource="http://wiki.opaals.org/TUT"/>
      <foaf:interest>
         <foaf:Document rdf:about="http://wiki.opaals.org/Information Visualisation">
            <dc:title>Information
Visualisation</dc:title>
            <dc:description>Existing wiki page</dc:description>
            <dc:identifier rdf:resource="http://wiki.opaals.org/Information
Visualisation"/>
         </foaf:Document>
      </foaf:interest>
      <foaf:interest>
         <foaf:Document rdf:about="http://wiki.opaals.org/Social_networking">
            <dc:title>Social networking</dc:title>
            <dc:identifier rdf:resource="http://wiki.opaals.org/Social_networking"/>
            <dc:description>A concept</dc:description>
         </foaf:Document>
      </foaf:interest>
      <foaf:interest>
         <foaf:Document rdf:about="http://wiki.opaals.org/Social_Network_Analysis">
            <dc:title>Social Network Analysis</dc:title>
            <dc:identifier
rdf:resource="http://wiki.opaals.org/Social_Network_Analysis"/>
            <dc:description>A concept</dc:description>
         </foaf:Document>
      </foaf:interest>
      <foaf:interest>
         <foaf:Document rdf:about="http://wiki.opaals.org/Digital_Ecosystems">
            <dc:title>Digital Ecosystems</dc:title>
            <dc:identifier rdf:resource="http://wiki.opaals.org/Digital_Ecosystems"/>
            <dc:description>A concept</dc:description>
         </foaf:Document>
      </foaf:interest>
      <foaf:interest>
         <foaf:Document rdf:about="http://wiki.opaals.org/Adaptive_hypermedia">
            <dc:title>Adaptive hypermedia</dc:title>
            <dc:identifier rdf:resource="http://wiki.opaals.org/Adaptive_hypermedia"/>
            <dc:description>A concept</dc:description>
         </foaf:Document>
      </foaf:interest>
      <foaf:interest>
         <foaf:Document rdf:about="http://wiki.opaals.org/Open_source">
            <dc:title>Open source</dc:title>
            <dc:identifier rdf:resource="http://wiki.opaals.org/Open_source"/>
            <dc:description>A concept</dc:description>
         </foaf:Document>
      </foaf:interest>
      <foaf:interest>
         <foaf:Document
rdf:about="http://wiki.opaals.org/Software_development_methods">
            <dc:title>Software development methods</dc:title>
            <dc:identifier
rdf:resource="http://wiki.opaals.org/Software_development_methods"/>
            <dc:description>A concept</dc:description>
         </foaf:Document>
      </foaf:interest>
```

```
      <foaf:homepage rdf:resource="http://matriisi.ee.tut.fi/hypermedia/"/>
      <foaf:publications rdf:resource="http://trip.cc.tut.fi/julkaisut/english.html"/>
      <foaf:holdsAccount>
         <foaf:OnlineAccount>
            <rdf:type rdf:resource="http://opaals.org/draft/workspace/wiki"/>
            <foaf:accountServiceHomepage rdf:resource="http://wiki.opaals.org/"/>
            <foaf:accountName>huhtis</foaf:accountName>
         </foaf:OnlineAccount>
      </foaf:holdsAccount>
   </foaf:Person>
```

Listing 5.1.1.2 *Example of OPAALS 2008 delegate data*

```
<delegate id="huhtamaki">
         <firstname>Jukka</firstname>
         <lastname>Huhtamäki</lastname>
         <email>jukka.huhtamaki@tut.fi</email>
         <organisation>
            <name>Hypermedia Laboratory of Tampere University of Technology</name>
         </organisation>
         <city>Tampere</city>
         <country>Finland</country>
         <address xmlns:kml="http://earth.google.com/kml/2.0" type="google.geoinfo">
            <name>Tampere, Finland</name>
            <lat>61.4980229</lat>
            <lng>23.7648591</lng>
            <kml xmlns="http://earth.google.com/kml/2.0">
               <Response>
                  <name>Tampere, Finland</name>
                  <Status>
                     <code>200</code>
                     <request>geocode</request>
                  </Status>
                  <Placemark id="p1">
                     <address>Tampere, Finland</address>
                     <AddressDetails xmlns="urn:oasis:names:tc:ciq:xsdschema:xAL:2.0"
Accuracy="4">
                        <Country>
                           <CountryNameCode>FI</CountryNameCode>
                           <CountryName>Finland</CountryName>
                           <AdministrativeArea>
                              <AdministrativeAreaName>Western
Finland</AdministrativeAreaName>
                              <SubAdministrativeArea>
                                 <SubAdministrativeAreaName>Pirkanmaa</SubAdministrati
eAreaName>
                                 <Locality>
                                    <LocalityName>Tampere</LocalityName>
                                 </Locality>
                              </SubAdministrativeArea>
                           </AdministrativeArea>
                        </Country>
                     </AddressDetails>
                     <Point>
                        <coordinates>23.7648591,61.4980229,0</coordinates>
                     </Point>
                  </Placemark>
               </Response>
            </kml>
         </address>
         <image uri="http://wiki.opaals.org/huhtis/FactsAboutPerson?
ction=AttachFile&amp;do=get&amp;target=opaals_fact_person_image.jpg"/>
         <!-- List of interests from the MindTrek registration form →
         <interestlist>
            <interest>community networks</interest>
```

```
            <interest>community visualisation</interest>
            <interest>data processing pipelines</interest>
            <interest>digital ecosystems</interest>
            <interest>social network analysis</interest>
            <interest>socio-technical perspective</interest>
         </interestlist>
         <opaals:profile accountname="huhtis">
            <foaf:Person xmlns:ws="http://opaals.org/draft/workspace/"
                         rdf:about="mailto:huhtis@opaals.org">
               <foaf:img rdf:resource="http://wiki.opaals.org/huhtis/FactsAboutPerson?
action=AttachFile&amp;do=get&amp;target=opaals_fact_person_image.jpg"/>
               <foaf:name>Jukka Huhtamäki</foaf:name>
               <foaf:mbox rdf:resource="mailto:jukka.huhtamaki@tut.fi"/>
               <foaf:Organization rdf:resource="http://wiki.opaals.org/TUT"/>
               <foaf:interest>
                  <foaf:Document rdf:about="http://wiki.opaals.org/Information
Visualisation">
                     <dc:title>information
visualisation</dc:title>
                     <dc:description>Existing wiki page</dc:description>
                     <dc:identifier rdf:resource="http://wiki.opaals.org/Information
Visualisation"/>
                  </foaf:Document>
               </foaf:interest>
               <foaf:interest>
                  <foaf:Document rdf:about="http://wiki.opaals.org/Social_networking">
                     <dc:title>social networking</dc:title>
                     <dc:identifier
 rdf:resource="http://wiki.opaals.org/Social_networking"/>
                     <dc:description>A concept</dc:description>
                  </foaf:Document>
               </foaf:interest>
               <foaf:interest>
                  <foaf:Document
rdf:about="http://wiki.opaals.org/Social_Network_Analysis">
                     <dc:title>social network analysis</dc:title>
                     <dc:identifier
 rdf:resource="http://wiki.opaals.org/Social_Network_Analysis"/>
                     <dc:description>A concept</dc:description>
                  </foaf:Document>
               </foaf:interest>
               <foaf:interest>
                  <foaf:Document
rdf:about="http://wiki.opaals.org/Digital_Ecosystems">
                     <dc:title>digital ecosystems</dc:title>
                     <dc:identifier
rdf:resource="http://wiki.opaals.org/Digital_Ecosystems"/>
                     <dc:description>A concept</dc:description>
                  </foaf:Document>
               </foaf:interest>
               <foaf:interest>
                  <foaf:Document
 rdf:about="http://wiki.opaals.org/Adaptive_hypermedia">
                     <dc:title>adaptive hypermedia</dc:title>
                     <dc:identifier
rdf:resource="http://wiki.opaals.org/Adaptive_hypermedia"/>
                     <dc:description>A concept</dc:description>
                  </foaf:Document>
               </foaf:interest>
               <foaf:interest>
                  <foaf:Document rdf:about="http://wiki.opaals.org/Open_source">
                     <dc:title>open source</dc:title>
                     <dc:identifier
rdf:resource="http://wiki.opaals.org/Open_source"/>
                     <dc:description>A concept</dc:description>
                  </foaf:Document>
               </foaf:interest>
               <foaf:interest>
```

```
                  <foaf:Document
rdf:about="http://wiki.opaals.org/Software_development_methods">
                     <dc:title>software development methods</dc:title>
                   <dc:identifier
rdf:resource="http://wiki.opaals.org/Software_development_methods"/>
                     <dc:description>A concept</dc:description>
                  </foaf:Document>
               </foaf:interest>
               <foaf:homepage rdf:resource="http://matriisi.ee.tut.fi/hypermedia/"/>
               <foaf:publications
rdf:resource="http://trip.cc.tut.fi/julkaisut/english.html"/>
               <foaf:holdsAccount>
                  <foaf:OnlineAccount>
                     <rdf:type rdf:resource="http://opaals.org/draft/workspace/wiki"/>
                     <foaf:accountServiceHomepage
rdf:resource="http://wiki.opaals.org/"/>
                     <foaf:accountName>huhtis</foaf:accountName>
                  </foaf:OnlineAccount>
               </foaf:holdsAccount>
            </foaf:Person>
         </opaals:profile>
      </delegate>
```

Appendix C: Wille2 HTTP Benchmarking

Description

In order to test the applicability of HTTP to Wille service transactions, and verify that our implementation of Wille server and HTTP service publication, several benchmarking tests were performed with wille2. The tests for HTTP benchmarking were 1) *multiple local servers benchmark*, 2) *HTTP upload size benchmark* and 3) *HTTP timeout benchmark*.

All tests were performed on locally on a single machine, with both server and client processes running in parallel. All tests were performed on a PC system with a 1.86 GHz single core processor, 1.00 GB of memory. Before running each tests, CPU was verified to be at least 95 percent idle, and amount of free memory to be at least 75 percent. System was running Windows XP (32-bit) operating system, with Python, Wille2 and Java run-time environments installed.

In each benchmarking test, a different aspect of Wille HTTP server function was tested. Full source code for each benchmark test can be found from Wille 2 Bundle at `wille.tests` package. The tests reported here performed benchmarking as follows:

- In *multiple local servers benchmark* (`wille.tests.multiple_servers`), we tested maximum number of Wille servers that could be set up and ran in parallel
- In *HTTP timeout benchmark* (`wille.tests.http_timeout`), we generated a series of increasingly long HTTP requests to test how long a server can have a request in processing, without dropping it or before any timeouts occur
- In *HTTP upload size benchmark* (`wille.tests.http_upload_size`), we generated a series of increasingly big HTTP requests to test for effiency of data transfer and for the maximum size for input data in an HTTP POST request.

When considered significant, execution time and memory usage were measured. In HTTP upload size benchmark, we also calculated a theoretical transfer rate. Note that the test setup considered a single machine running all processes locally. Hence, the results do not have qualities present in real network that may reduce quality of service or restrict transfer rate.

Results

As the key results of the benchmark testing, we could verify that Wille's HTTP services implementation works within a given limit, and could identify potential limitations and bottlenecks.

In *multiple local servers benchmark*, we could successfully set up up to 90 Wille server instances and perform service search (by service pooltype, name and type) from the pool of all the services. When setting up more than 90 server instances, unhandled exceptions occured. Trying to set up more than 150 servers resulted in socket error (`socket.error: (10013, 'Permission denied')`)

In *HTTP timeout benchmark*, a series of increasingly long HTTP requests (without minimal payload) were generated. Starting from 1 second, we could increase length of a request up to 5 minutes without getting any dropped requests or timeouts. We predict that even longer requests would be possible in our local machine only setting.

In *HTTP upload size benchmark* we transferred a payload (consisting of random bytes of data), starting from 1 megabyte. As summarised in figure 1., We slowly increased the size of data up to 125 megabytes. Up until 125 megabytes, both execution time and memory usage kept steadily rising. However, with uploads larger than 25 megabytes, transfer rate began to decline: in comparison to 1 megabyte upload, at 125 megabytes roughly around 50% transfer rate could be achieved.

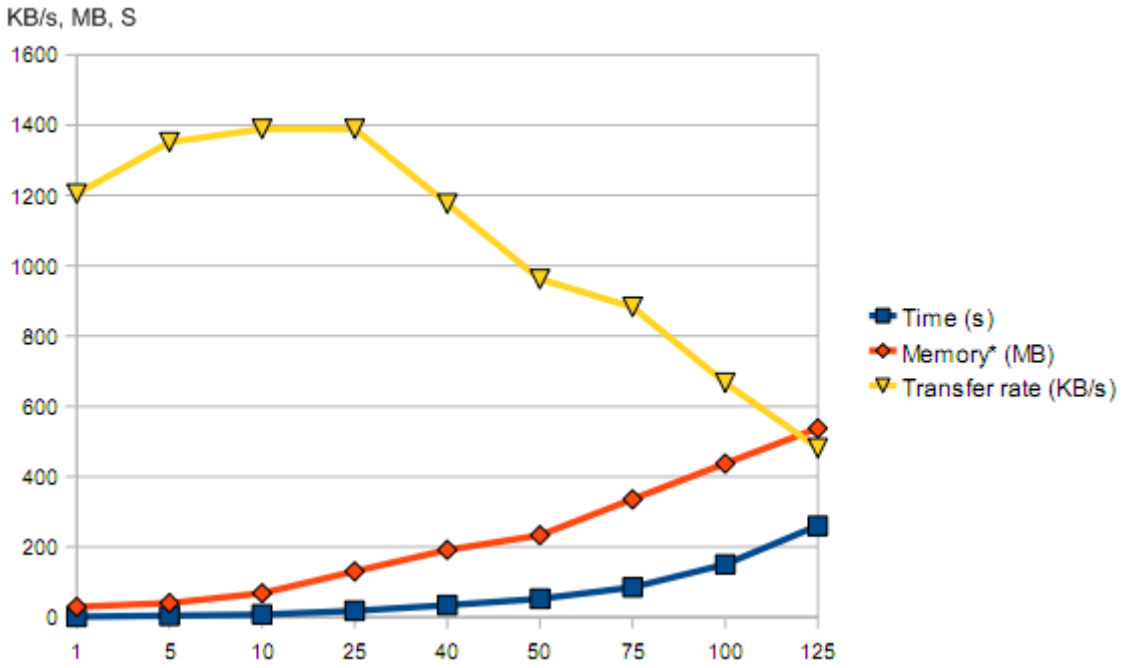


Figure 1. Results from HTTP upload size benchmark

Conclusions

Based on HTTP benchmark tests, we see that - on our implementation - Wille services can be fairly reliably executed on local machine or local machine-like conditions, given that request payload and request length are limited. We could not find a specific maximum for request size or length. However, due to inherent unreliability of networking, we do not recommend using Wille server for very long running requests (let's say more than 2-3 minutes) or for transferring over 100 megabytes of data in a single request. It would be unrealistic to assume that HTTP timeouts would not occur in real-life setting: even a temporary disconnection from the network will cause HTTP request to time out. Also, while even very long running service requests are possible with Wille, reliability is not guaranteed. For reliable, long-running service executions or transfer of large data objects (>100 megabytes) more sophisticated mechanisms should be introduced.