Cocytus: parallel NLP over disparate data

Noah Evans — Masayuki Asahara — Yuji Matsumoto

Nara Institute of Science and Technology
8916-5, Takayama-cho, Ikoma-shi
Nara 630-0192 JAPAN

ABSTRACT. *As NLP deals with larger datasets and more computationally expensive algorithms, cutting–edge NLP research is increasingly becoming the province of companies like Google who can use an astronomical amount of resources to do NLP tasks. Smaller institutions are being left behind. In addition to this lack of resources, what resources a typical researcher does have access to are represented in a variety of differing, incompatible data formats and operating system semantics. NLP researchers devote a large amount of research time developing NLP tools to support a variety of different data formats, time that could be spent doing productive research. To solve these problems of data representation and processing huge data, this paper presents Cocytus, a platform for creating NLP tools loosely based on Unix, that handles different data formats and parallel computation transparently, thus allowing institutions to make maximum use of their resources.*

RÉSUMÉ. *Au fur et à mesure que le TAL se confronte à des données de plus en plus volumineuses, et fait appel à des algorithmes de plus en plus complexes, la recherche en TAL devient de plus en plus l'apanage de sociétés telles que Google, qui peuvent utiliser une quantité astronomique de ressources pour accomplir des tâches de TAL. Les instituts de recherche les plus petits ne peuvent pas suivre. Les ressources auxquelles les chercheurs ont accès ne sont pas seulement insuffisantes, elles sont également très hétérogènes, se présentant dans des formats très variés, incompatibles entre eux. Les chercheurs en TAL consacrent donc une part importante de leur temps à développer des outils qui supportent cette variété de formats, au détriment d'activités plus productives. Cocytus, la plateforme présentée dans cet article, est précisément conçue pour apporter des solutions à ces problèmes d'hétérogénéité et de volume des données à traiter. Cocytus permet de développer des outils de TAL sur la base d'utilitaires Unix, offrant un accès transparent aux données et à des ressources de calcul parallèles. Cocytus permet ainsi aux instituts de recherche de tirer le meilleur parti de leurs ressources.*

KEYWORDS: *querying, scalable, cloud computing, Inferno, MapReduce*

MOTS-CLÉS : *passage à l'échelle, informatique distribuée, Inferno, MapReduce*

Traitement Automatique des Langues. Volume 49 – n° 2/2008, pages 271 à 293

1. Motivation

NLP applications are still catching up to the explosive growth of data sources like the web. The size of corpora are no longer measured in megabytes or gigabytes —now terabytes and petabytes— and the amount of this data is increasing exponentially.

In spite of this increase, most NLP applications are not designed to handle large datasets. They are one-off tools designed to solve a particular problem or prove a research point. These applications deal with data in an ad hoc manner, emphasizing quick solutions over generality. This lack of standardization, as well as rapid progress and differentiation in areas such as character sets and structured data, makes developing a viable NLP tool difficult without wasting effort developing support for a variety of differing data formats and character sets.

In addition to this wasted effort, NLP algorithms are typically computationally intensive, especially statistical NLP algorithms. NLP computations, especially machine learning algorithms like classifiers, can take weeks to complete even on fast machines. This intensity, coupled with the massive size of datasets available, makes handling the current problems facing NLP difficult to solve with common tools. The ad hoc implementation of most tools makes it difficult for them to handle larger sets of data.

Currently there are two common methods used to solve the problem of NLP scalability. The first and, until recently, the most common approach has been to use efficient specialized algorithms and formats on top of increasingly more powerful systems. By increasing the efficiency of the solution and the computational power available to implement the solution, it has been possible to keep up with the increasing amount of linguistic data. The move to XML databases and specialized querying described in Bird *et al.* (2004) is indicative of this approach to dealing with NLP problems scalably.

The other, more recent, approach is a move towards cloud computing, dealing with huge amounts of data by distributing data over a variety of computing resources. Google is the most successful advocate of this approach, processing over 20 petabytes of data daily using its MapReduce algorithm (Dean and Ghemawat, 2008).

However, as the size of data increases, smaller institutions, notably small universities and startup companies, are being left behind. Even small cloud clusters can be beyond the resources of a small institution. There are attempts to make high–performance NLP problem–solving widely available using MapReduce (Pantel, n.d.), but these systems rely on an external system shared among institutions.

2. Cocytus: a NLP framework based on Inferno

The Cocytus system attempts to mitigate this comparative lack of resources by providing an architecture that maximizes *all* of the available resources of an institution to deal with NLP problems, especially unused computational resources like lab computers and linguistic resources like corpora in a variety of formats. With this goal

in mind Cocytus has a structure that is very different from most conventional NLP systems. Other systems like UIMA (Ferrucci and Lally, 2004) or GATE (Cunningham *et al.*, 1995) are typically built on top of an operating system with very little integration with the system itself. They are self–contained, heavily engineered systems that take a variety of NLP problems and workflows into account and create workflow management architectures to solve NLP problems.

Cocytus does not take this approach. Instead of forcing the user into a predefined workflow pattern or defining the system as a monolithic program, Cocytus is built on top of Inferno (Dorward *et al.*, 1997), a hosted, distributed operating system developed by Bell Labs and now distributed and maintained by Vita Nuova[1]. Inferno gives users greater flexibility by providing a traditional Unix-style interface that deals with issues such as supporting a variety of character sets or handling a variety of structured data formats, that traditional systems deal with at the software level, *at the systems level*. Cocytus supports data formats and parallelization not in a monolithic system, but in a variety of small programs that alter the behavior of the operating system itself, presenting a variety of disparate resources in a standardized way for all programs using Cocytus.

This allows users to take advantage of the facilities and tools provided by the underlying operating system to solve NLP problems using a workflow and development cycle similar to Unix. Inferno's architecture allows Cocytus to be small (Cocytus is less than 10,000 lines of code, not counting its underlying operating system, Inferno, and most of Cocytus is prebuilt modules to deal with various character sets and structured data formats). Its small size and resulting comprehensibility encourage further research by letting users change and improve all of the system.

Currently Cocytus is being used to solve a variety of problems in the NAIST computational linguistics lab, primarily N-gram and structured data editing and extraction.

This paper is an overview of the system.

3. Foundations

The underlying structure of Cocytus differs substantially from current NLP middleware systems. The following section describes aspects of the underlying architecture of Cocytus necessary to understand the implementation and advantages of the system.

Cocytus implements this system by using a hosted operating system, Inferno, and its native support for pipes and UTF-8, to create a system that allows the creation of NLP tools in a modular way, with a consistent way of representing NLP data. Cocytus implements this system by changing the representation of linguistic resources —currently limited to corpora— creating pipelined applications and by distributing

1. `http://www.vitanuova.com`.

the now modular applications computation over a heterogeneous set of computing resources. Figure 1 gives an overview of the system and its structure:

Figure 1. *Cocytus is an NLP platform that allows researchers to solve NLP problems in a scalable, reusable way. Instead of trying to implement its functionality as a software program running in user space, Cocytus forces all of the dependencies of a traditional NLP platform —tools to deal with different data formats and workflow execution— into the behavior of the operating system itself. This allows NLP researchers to concentrate on creating tools to solve problems, not deal with extraneous issues like concurrency and data formats.*

To implement this transparency of representation, Cocytus is fundamentally two structures built on top of Inferno. The first, the set of *conversion tools*, converts a variety of NLP data formats into Cocytus' native formats without any action by the users. This allows users to use a variety of linguistic resources, for example data marked up in XML and s-expressions, as the same format. This takes the complexity of format support out of tool development and moves it to the systems level, meaning that once a format is supported, it can be ignored by the user, as everything appears to be in one format. Cocytus puts special emphasis on representing character sets and structured data formats in a common format.

Cocytus' other structure is a shell–based *process distribution* mechanism. This allows Cocytus to take preexisting tools and distribute them transparently over a variety of machines with no explicit distribution by the user. This allows Cocytus users to tackle problems and tasks that are too processor –or resource– intensive for traditional systems.

3.1. *Inferno*

Inferno is a Unix derivative by way of Plan 9 (Pike *et al.*, 1995), which leads to an operating system that functions similarly to Unix in many ways. Inferno follows the

basic Unix philosophy: input and output are through files, including device IO which is done through file interfaces. It contains a limited but powerful set of system calls, create, read, write, open, remove, stat, spawn(fork), and pipe among others. It also includes new versions of the standard set of Unix tools, a shell, systems programming language Limbo (Ritchie, n.d.), editor, window system, and other user programs.

Architecturally Inferno is primarily a hosted operating system, although the system can also be built as native kernel. Both kernels are virtual machines running a register–based virtual machine called DIS (Winterbottom and Pike, n.d.). Figure 2 gives an overview of the structure of an Inferno system:

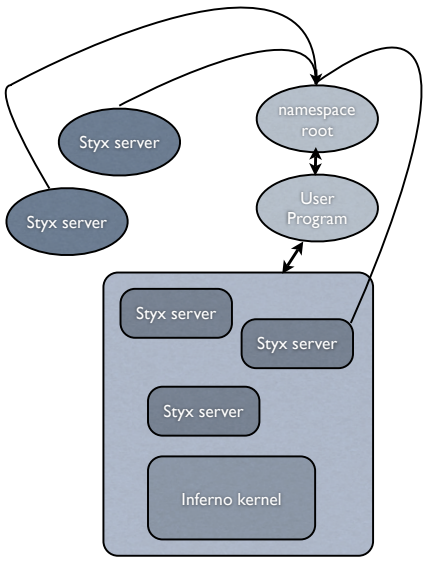

Figure 2. *The Inferno operating system is made up of Styx servers which serve namespaces, Inferno's version of filesystems. Unlike other systems namespaces are not handled by the kernel, they are handled by individual processes. This allows each process to make its own view of the system. Processes can share, replace, and create resources transparently, including over networks.*

3.1.1. *Namespaces*

While Inferno's Unix ancestry gives it a simple consistent interface, many of its advantages come from where it differs from Unix. What makes Inferno unique in relation to Unix is its novel interpretation of file systems. Inferno does not follow the typical Unix model, where file systems are a representation of the underlying physical files of the system with a few additional unique files synthesized by the kernel.[2] In-

2. E.g., /proc or /dev.

stead, Inferno has per process *namespaces*, synthetic filesystems that act in a similar way to Unix mount tables (Thompson, 1978), but whose tables are maintained by individual processes, not by the kernel. This allows processes to dynamically determine their environment, adding, removing, and overriding resources and file hierarchies as they see fit, including devices like monitors, mice, and network devices as well as completely abstract file systems like file–based representations of mail queues.[3] This allows systems to easily represent, distribute, and replace *any* system resource using the *Styx* protocol described in the next section. Figure 3 describes an Inferno namespace.

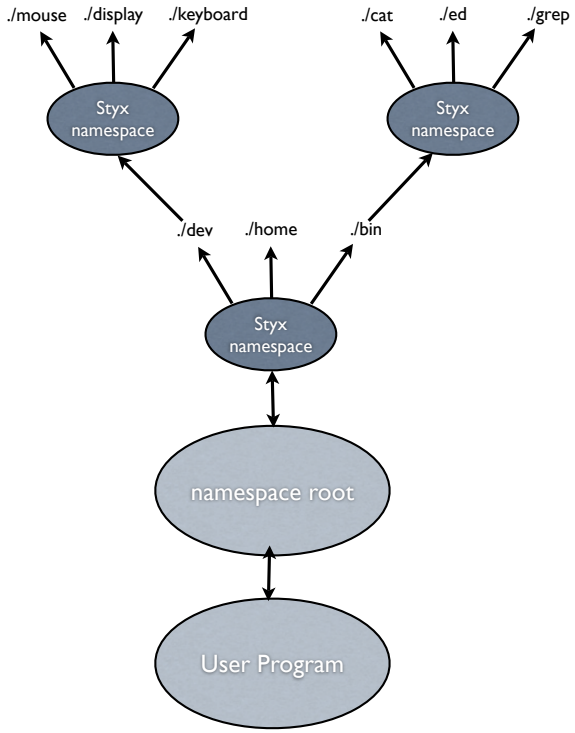

Figure 3. *The program namespace is determined through the root, then the namespace is made synthetically by combining the resources together by mounting them in other nodes of the namespace. New programs inherit their parents' namespace and any changes made to the program's namespace are private to that program and its children. Changes made to one program's namespace are invisible to other programs, allowing different programs to have namespaces that are completely different from one another.*

3. Plan 9's upasfs(1) is a good example of this kind of file system.

3.1.2. *Distributing data and services using filesystems*

Inferno distributes data and resources over networks with a protocol called Styx (Pike and Ritchie, 1999).

At its most basic, Styx is a distributed file system protocol, describing file systems calls instead of the kernel. This allows the kernel to avoid hard-coding *any* file systems calls inside the kernel itself; it just provides an interface for drivers which speak Styx. The native filesystem, the system devices, the process table, all of these are implemented not as part of the kernel but as optional device drivers that can be statically linked into the kernel at compile time. These drivers can then be added to the user–viewable namespace as necessary, by mounting the unique representation of the device to the current namespace.

Because Styx is a protocol, not a format, the system can share any part of the namespace transparently over a network: all that is necessary is a networked device that can read and write the Styx protocol (e.g., Inferno or Linux machines with the proper kernel module) and understands Inferno's authentication mechanism. This protocol–based approach allows local and remote devices to be handled equivalently: any device is a file, and any file can be served over the network, meaning any device or service can be shared transparently without the need for Remote Procedure Calls (RPCs) like XML-RPC (Winer, n.d.) or application–specific device sharing tools, like USB over IP (Hirofuchi *et al.*, 2005).

One important result of this approach and especially important to the design of Cocytus is cpu(1), a tool for distributing computation over Inferno nodes. cpu(1) does not need a special protocol or platform to distribute computation, it just reorganizes the user's namespace in order to present remote computation locally. It takes local parts of the user's namespace, for example the keyboard, mouse, and monitor, and merges them with those of a remote system (Cox *et al.*, 2002), unlike ssh (Ylonen *et al.*, 2002) or other systems which just implement a variety of remote shell and file transfer protocols. Because cpu(1) is a combination of namespaces and not a protocol, commands are executed normally, by writing to the external console or via tools that multiplex the terminal. Cocytus makes extensive use of this distributability in distributing computation in its MapReduce implementation described in Section 6. Figure 4 shows how Styx namespaces can be shared.

4. Development in Cocytus

Tools developed in Cocytus are easier to implement and easier to reuse in new tools than tools developed in traditional systems. Tools developed in Cocytus combine UTF-8, pipelines, and a special format for structured data, TreePaths, to provide a system where the user can implement tools for individual tools separately and make them communicate over standard interfaces without the need for programming language–specific integration.

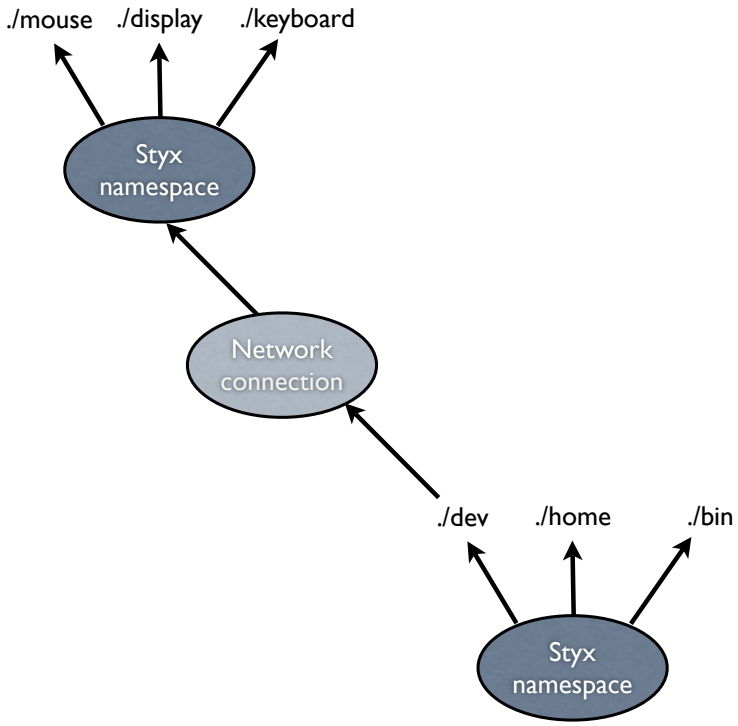

Figure 4. *Styx namespaces can be easily shared over a network. Since namespace interactions are a protocol rather than a facility provided by the system, as long as there is a transport mechanism that provides two-way communication between the host and server, it is possible for Styx namespaces to be shared transparently over a network.*

4.1. *NLP Workflows*

An NLP workflow is typically a process of adding annotation, a series of transformations starting with raw text and resulting in linguistic data containing sufficient information to solve a given NLP problem. For example, a morphological analyzer transforms raw text into tokenized words with morphological information attached. Another application, typically a dependency or syntactic parser, can then use that morphological information to establish the hierarchical structure of the linguistic data, describing the relationship between words and clauses. This process of annotation continues until there is sufficient information contained in the annotation to understand and solve some aspect of language processing, e.g., anaphora resolution, biographical summarization, or machine translation.

However, the ease of implementation of this workflow is hampered by a variety of issues. Text comes in different character sets; annotation is presented in different formats. The process of creating NLP tools is not one of implementing tool functionality so much as a matter of choosing, defining, or integrating different formats.

Even using common formats, communicating between applications is difficult. NLP annotation maps well to Unix pipes, whose stream–based approach models the staged, one-way process of adding successive layers of annotation. However, few NLP tools take advantage of this process. Take the NLP tools Chasen (Matsumoto *et al.*, 1997) and Cabocha (Kudo and Matsumoto, 2002), a morphological analyzer and dependency parser developed at Nara Institute of Science and Technology. Cabocha uses the output of Chasen as its input for creating dependency parses, an ideal application of pipes—the information about each sentence provided by the morphological analyzer is used as input for the dependency parser. However, Cabocha does this by integrating Chasen's core into the Cabocha executable instead. The differences between Unix and Windows methods of doing interprocess communication make using the pipeline model difficult and inconsistent across operating systems.

Cocytus, by using Inferno, eliminates this barrier to developing communicating systems by providing common semantics across all host systems. Tools developed in Cocytus can use of pipe–oriented development model because they will always be equivalent no matter what the platform.

Using Cocytus, the integration of separate applications such as Chasen and Cabocha is unnecessary. A Cocytus Chasen could just as easily output data to Cabocha over a pipe. Integration is unnecessary. A "Software Tools" (Kernighan and Plauger, 1976) approach, similar to the original Unix development model, becomes the standard way of interacting with the system.

4.2. *Formats for pipelining*

However, even if pipes are supported uniformly on the OS level, the majority of corpus data and other tools are not in the proper formats amenable to pipelined applications.

4.2.1. *UTF-8 as a format for NLP pipelines*

The character set of corpus format greatly affects the ability of that corpus to be used in pipelined applications.

Ascii–formatted text in English, like the Penn Treebank or the British National Corpus, works well in pipelined applications, but corpora in other languages, especially Asian languages like Japanese, have a variety of character sets capable of representing the same kind of language.

This creates a situation where any tool that wishes to support a specific language is forced to support a variety of different character sets within the tool itself, creating

development issues. Even worse, the character set may itself be impossible to pipeline. For example, any character set based on a multibyte format that requires a BOM (byte order mark) to determine its format will turn to gibberish on machines of differing endianness when used in a stream that can begin in arbitrary locations in a file. This is an especially big problem for systems that are meant to be used with a variety of distributed, heterogeneous computing resources like Cocytus.

Inferno, and thereby Cocytus, solves this problem by using UTF-8. Because UTF-8 is defined as a byte stream and covers most world languages, almost any language can be handled natively in Inferno. In addition, UTF-8 data is represented as 16 bit integers inside Inferno's virtual machine, solving the efficiency problems that arise when indexing strings of variable–width characters. These problems are traditionally cited when arguing against the use a byte–oriented formats like UTF-8. This allows any language on Inferno's virtual machine to handle UTF-8 naturally (string handling is part of DIS). Thompson (1993) details the previous issues with UTF-8 and gives a more detailed description of the creation of UTF-8 and its use in Plan 9 and, by extension, Inferno.

4.2.2. *Data formats and structured data representation*

The stream–based model of Cocytus is good for flat text strings without any dependencies, but handling structured data in a stream–based format is difficult. Cocytus solves the problem of structured data formats in streams by encoding the structure in the data format itself, much as Unix encodes the tree structure of the Unix filesystem in paths.

4.2.3. *Traditional NLP formats*

Traditionally, NLP data is formatted in recursive structures, like XML or s-expressions. Each of these formats requires complicated parsers which have a great deal of complexity and complicated semantics and implementations, especially facilities with a potential for ambiguity like XML namespaces. In addition, these formats are poorly suited to streaming. They require the system to keep track of the state of the stream between lines, leading to problems similar to those that arise dealing with wide characters without a BOM. The stream cannot start at arbitrary points, only at specific points where the state of the entire structure can be inferred (e.g., the root of an XML tree). Then the data must be parsed, represented, and then reencoded in the proper representation upon output. Figure 5 shows how this incompatibility makes it difficult to deal with traditional structured data formats using Unix tools.

4.2.4. *TreePaths*

To solve the problem of integrating the representation of structured data with Unix tools, we chose TreePaths (Evans *et al.*, 2007) as the primary structured data format for Cocytus.

```
% grep ADJP <<!
( (S
   (NP MEI Diversified Inc.)
   (VP agreed
       (S (NP *)
  to
  (VP acquire
      (NP LecTec Corp.)
      (PP for
  (NP (NP stock)
      (ADJP  currently
     worth
      (NP $
 17.6 million)))))))
  .)
!
      (ADJP  currently
```

Figure 5. *Applying a Unix tool, grep(1), to an s-expression. The grep command intends to extract an adjective phrase but, since the data is structured recursively, extracting the line containing the adjective phrase tag does not extract the entire phrase.*

Cocytus does not rely on the recursive methods most common in current NLP systems, using TreePaths, a format based on Unix paths, instead. TreePaths differ from traditional methods of representing structured data in NLP systems: they are depth–first enumerations of a tree, root to leaf, where each node is enumerated by its position relative to its siblings. Figure 6 shows a tree path.

This structure allows TreePaths to interact with pipeline–based tools instead of being queried and edited by format–specific or specialist tools. By encoding the location of any node in relation to the root explicitly, querying TreePath annotations becomes a matter of string matching, not parsing a recursive format. Subtree extractions based on data from parent nodes, i.e., extracting a subtree based on its parent node, can be done with one invocation of grep(1). Extractions based on the position or type of a child node in relation to its siblings can be done with two calls to grep(1) (one to get the child node, and one to extract all of its siblings). Finally, horizontal searches like the type provided by more recent languages such as LPath (Bird *et al.*, 2004) can be done using a simple stack–based extraction: use a stack to keep previous nodes, then, when two nodes satisfying the relationship are found, pop the stack. In–depth examples and a further explanation of TreePaths are described in Evans *et al.* (2007).

```
/1.S/1.NP/1.Continental
/1.S/1.NP/2.Airlines
/1.S/2.VP/1.is
/1.S/2.VP/2.NP/1.a
/1.S/2.VP/2.NP/2.unit
/1.S/2.VP/2.NP/3.PP/1.of
/1.S/2.VP/2.NP/3.PP/2.NP/1.NP/1.Texas
/1.S/2.VP/2.NP/3.PP/2.NP/1.NP/2.Air
/1.S/2.VP/2.NP/3.PP/2.NP/1.NP/3.Corp.
/1.S/2.VP/2.NP/3.PP/2.NP/2.,
/1.S/2.VP/2.NP/3.PP/2.NP/3.NP/1.Houston
```

Figure 6. *The context of a TreePath is encoded in the data structure itself like a Unix path. By encoding a depth–first enumeration from node to leaf in each line of the representation, vertical queries, subtree extraction, and editing become a matter of string matching. By pushing the lines of a TreePath onto a stack or doing two–pass processing, it becomes possible to do horizontal node querying as well.*

5. Data Representation

Even though Cocytus specifies a stream model, adhering to this model can be difficult for NLP researchers. Corpora come in different formats, many of which are cumbersome or incompatible with stream approaches.

This incompatibility compels any tool designer who wants to support a combination of these different formats to integrate support for these data formats into the tool itself. This requires constant reimplementation of format support when tool implementers attempt to write tools in an unsupported programming language or develop a new tool with a different interface.

Cocytus solves this representation problem by using namespaces to provide a unified view of disparate resources. This solution is based on the observation that, because Inferno namespaces are *synthetic*, they do not represent physical files or resources, just a server that responds to the Styx protocol, so developers can create *new* namespaces by implementing a Styx server to interact with an already existing resource. New namespaces can then be integrated into the current namespace in the same way as any other device.

The best example of this style of problem solving is trfs(4)[4], a tool for handling spaces in the host operating system's filenames in Inferno (Inferno considers spaces a token delimiter in file names). trfs(4) takes an existing resource, a host OS names-

4. See `http://groups.google.com/group/comp.os.plan9/browse_thread/thread/ec0a67f915b27887/cf0151e48598d25e?hl=en&lnk=st&q=trfs+nemo#cf0151e48598d25ei` for a discussion and the source code for trfs.

pace, and applies a transformation to it, converting their filenames to contain another character instead of spaces when their parent directories are read.

Using trfs as a model, we have developed a way of presenting disparate linguistic resources in a single, standard way. This technique relies on *namespace transformations* that take a variety of resources, different character sets, and structured data formats, and convert them to Cocytus' native formats. The system takes a set of linguistic resources in a file system and creates a new namespace using the resource filesystem as a guide, duplicating the names and the directory hierarchy of the files in a resource.

This approach allows developers to ignore the issues that arise from implementing support for various formats. By providing an alternative representation of different resources in Cocytus' native formats, foreign format support by user programs is unnecessary, and resources appear automatically in the standard format. Once the implementer chooses the standard data format, the system itself manages the supporting other formats, eliminating any need for programmers and users to handle other formats manually.

This format handling system is implemented as a series of *namespace servers* which take a previous namespace, the original or *intermediate namespaces* and perform a predefined transformation on the chosen namespace. In this manner, it is possible to perform a series of transformations on a namespace, with each intermediary namespace applying successive transformations, finally terminating in the desired format. Figure 7 shows an overview of the process.

5.1. *utfs and tpfs*

Currently Cocytus has two file modules, utfs and tpfs, to do conversions. The executables are actually wrappers around tmfs, calling the module and passing command line arguments to the system itself. The modules are then included in a fileserver tmfs which applies the module's transformation functions to the file system calls. These tools are described in greater depth in Evans (2007).

5.1.1. *tmfs*

Cocytus' conversion system is constructed around a central program, tmfs. tmfs is a namespace transformer, that takes the representation of a resource in one namespace, e.g., a text file or annotated data, and transforms how that representation is depicted. tmfs has the ability to change the name, contents, and hierarchy of any given namespace. Changes can be transparent in both directions, reading converting to a standard format and writing converting *back* to the original format.

This gives the conversion system the ability to take resources that are presented in two different but regular ways, for example, systems with different character sets or

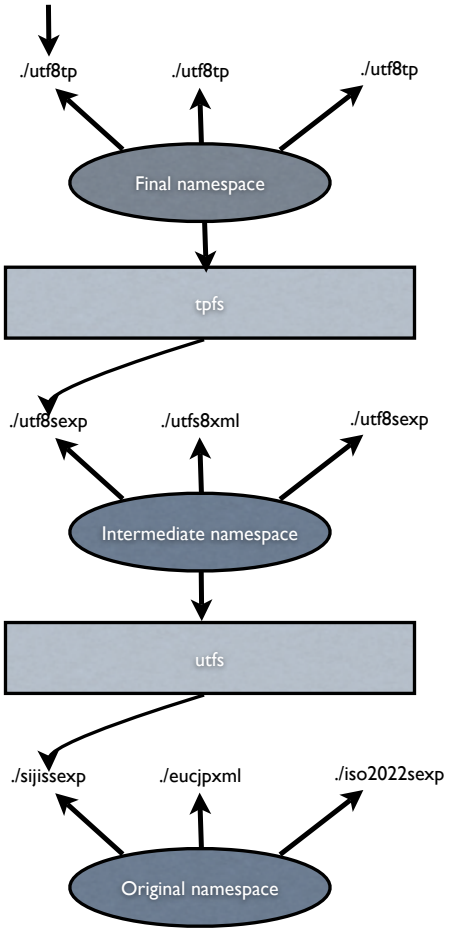

Figure 7. *tpfs and utfs allow the system to present all linguistic resources available to the researcher in standardized formats. utfs takes the original namespace, detects the character set of its files, and applies a conversion filter to convert the character set of the original files to UTF-8 when they are read or written. This intermediate transformed namespace is then transformed again, this time using tpfs, which uses a frequency heuristic to determine the structured data type. tpfs then parses the structured data and converts it to TreePaths, Cocytus' native structured data format. The final namespace that results from both of these transformations represents all of the available resources uniformly as UTF-8 and TreePaths.*

differing types of structured data, and change the representation of those resources to a common format.

tmfs performs these conversions by acting as a Styx transformer. It takes Styx calls from a namespace and then applies a specified function to the call. This is analogous to the Unix process of *stream editing*, sending data over a pipe, applying a transformation and then using the output for different purposes. The Unix awk(1) and sed(1) commands are commonly used for stream editing.

However, tmfs does not define the transformations itself; it defines the *interface* to a transformation module instead. The user, or more often the system administrator, provides an external module that provides the relevant transformations. These transformation modules form the basis of resource representation in Cocytus, allowing it to provide a uniform representation of disparate resources. Currently the system contains two modules, utfs and tpfs, to provide common representations of character sets and structured data.

5.1.2. *utfs*

utfs is a module for tmfs, intended to take a variety of character sets and convert them to UTF-8 when a namespace containing files with non-UTF-8 character sets is read and written. It does this by caching reads and writes and then recognizing the character set of the original file using an Inferno port of the Mozilla character set detection algorithm (Li and Momoi, 2001). Once the character set is detected, it applies one of Inferno's tcs(1) suite of character conversion modules and converts character sets to the proper types.

5.1.3. *tpfs*

tpfs takes a namespace containing structured data formats, currently XML and s-expressions, and expresses them as TreePaths. tpfs does this by treating all files as text files and then using the frequency distribution of characters in the files to determine their format. Once the format is determined when the files are read and written, the data is parsed into a cache and converted. This caching is necessary because TreePaths require shorter reads (TreePath nodes are self-contained) compared to other formats. Only when structured data is completely parsed and converted can it be made available, hence the need for caching.

6. Maximizing computation

Cocytus' development model and way of representing resources provide the common foundation Cocytus uses to maximize the computing resources available to an institution.

By maintaining data in a common format that is processed identically regardless of architecture, operating system, and data format, all of the resources of a research lab, lab machines of varying architectures (e.g. Sparc, 386, and PPC), operating systems (Mac, Solaris, Windows, Linux), and corpora in different languages can all be used as part of a Cocytus installation in a unified way.

Using this unified model of resource representation, we have developed a subsystem of Cocytus that handles process distribution, parallelizing NLP computations transparently. This system uses a MapReduce implementation built on top of the shell to distribute computation. Because the smallest unit of Cocytus is a pipelined program, not a module, Cocytus' MapReduce implementation distributes traditional Inferno (Unix) commands instead of distributing modules, thus allowing any user program to be parallelized, so long as it follows the Cocytus pipeline model. This process differs from traditional MapReduce systems which use modules or objects in a language like C++ or Java to set up computation (Bialecki *et al.*, n.d.).

Using the shell to distribute processes gives Cocytus an advantage over traditional MapReduce systems. Modules do not have to be written for MapReduce specifically, traditional Unix tools can be used as mappers and reducers, eliminating the need to create specific tools. Instead of parallelizing objects or modules, we are parallelizing shell commands. This is what gives Cocytus its development model. Users can develop tools using toy data quickly, then once the solution is deemed sufficient it can be distributed over large datasets. This helps groups with limited resources avoid wasting resources on potentially mistaken results—the tools used in the MapReduce have been debugged already. Figure 8 gives an overview of Cocytus' process distribution.

6.1. *Performance*

NLP tools are typically evaluated by their processing–time efficiency. The goal of any system is to return the user's desired output in the shortest amount of time possible. The same is true of NLP platforms.

In theory, Cocytus' method of doing NLP problem–solving, using un-optimized, stream–oriented processing of NLP data, makes Cocytus slower than other NLP systems which are more heavily optimized for specific problem domains. In practice, we expect that this lack of performance is inconsequential using Cocytus' development model. Tools are developed using small datasets which —even using linear, text–based methods— are sufficiently responsive to user input. In the case of large datasets, Cocytus performs better than optimized methods: Cocytus' line–oriented IO model and its ease of distributing computation using namespaces allows Cocytus to overcome its slowness using brute force methods.

With these expectations in mind, we tested the system, extracting noun phrases using the following resources. We compared the system against a common optimized querying tool, tgrep2 (Rohde, 2004) and, because we do not have a high–performance Lpath querying system[5], we extrapolated the speed of Lpath based on Bird *et al.* (2004)'s results compared to tgrep2.

5. An Lpath implementation exists for NLTK (Loper and Bird, 2002), a system for teaching NLP programming, but it is not optimized for performance, so it was not measured here.

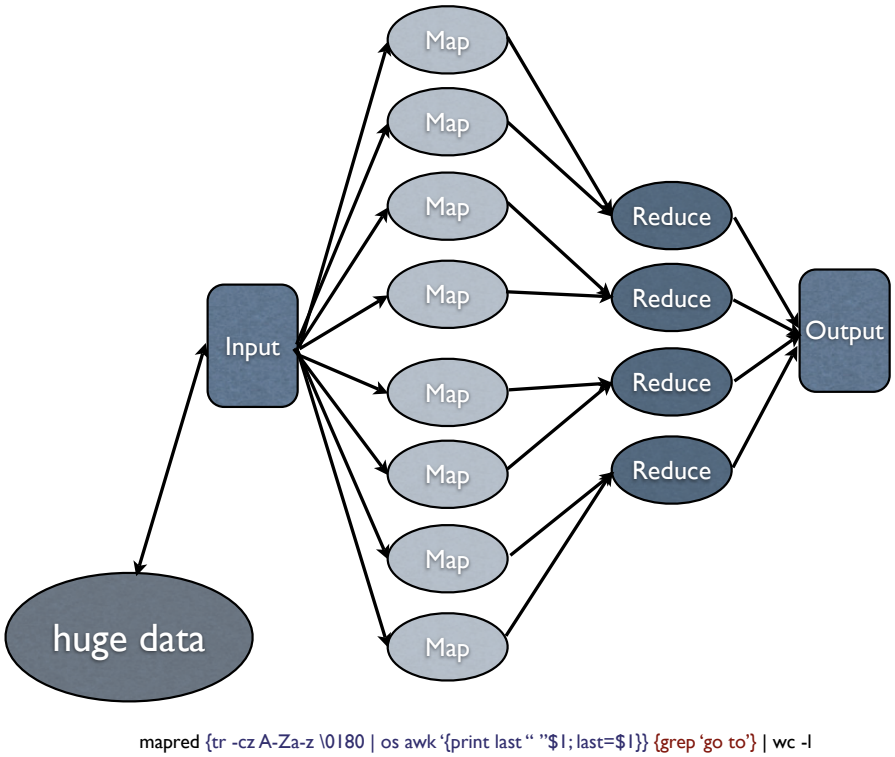

Figure 8. *Cocytus' MapReduce system is built on top of the shell. It has a special command for aggregating and distributing data. Process distribution, both maps and reduces are handled transparently using Inferno's cpu(1) command, which sends the proper shell command and options to the remote machine. Output is sent to the user's standard output. If the user chooses to, they can redirect the output from the shell.*

We tested the data on the PennTreebank, represented as tgrep2's special index and TreePaths respectively. The indexed version took place on a Sun Fire V1280 server with 96 gigabytes of RAM and a 64 node grid of dual processor 1.8ghz AMD Opteron servers with 2 gigabytes of RAM each to test Cocytus.

The table is logarithmic, comparing doubling of data size with the corresponding logarithmic scale measure of cpu time. Figure 9 shows the results for a variety of queries. Because there is not enough Penn Treebank data to compare to truly huge data–sets, the Penn Treebank was concatenated onto itself. The X axis scale based on n is the number of complete sets of the Penn Treebank searched. Figure 10 shows these results.

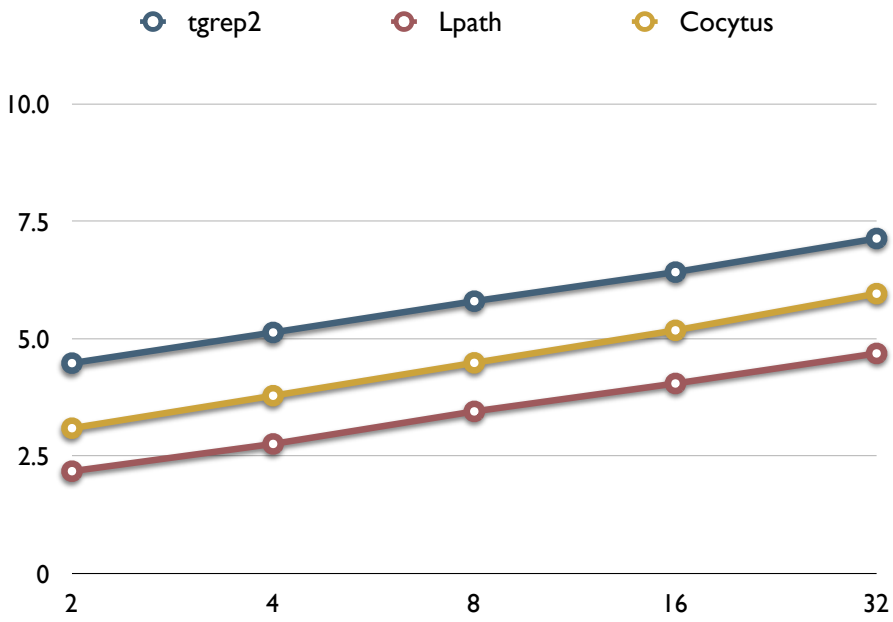

Figure 9. *Cocytus' performance. Cocytus, by virtue of its being able to distribute NLP computations transparently, remains competitive with other NLP systems, despite its simplicity and lack of optimization.*

6.2. *Programmer efficiency*

As well as measuring computational efficiency, another important measure of the success of Cocytus is the extent it simplifies the NLP tool development process by eliminating code that would have been written to support different datasets and optimize efficiency.

In many cases using Cocytus eliminates the need to write new NLP tools at all. Querying can be done with creative uses of grep(1), simple tokenization and concordancing can be done with creative combinations of tr(1), sort(1) and uniq(1) (Church, 1994)

However, in fairness to other systems and their tools, we took the main tool used for querying and extracting in Cocytus, grep(1), and compared its length in lines of code to two other tools, tgrep2(1) and Chasen. While Chasen may seem an odd choice, —a morphological analyzer compared to two extraction tools, Chasen is relevant to the current comparison because roughly $1/6$ of Chasen is devoted to Japanese character set handling.

Figure 10 shows a comparision between tgrep2, Inferno's grep and Chasen's character set handling code (Chasen itself is 4971 lines of C code). The code metric was measured using c_count(1)[6].

The relative size of each tool is instructive. If Cocytus had tried to mimic the development models for NLP tools, the developer of grep(1) would have had to write more than twice as much code *just to support Japanese character sets*. Cocytus, by moving character set handling to the systems level, avoids these problems. This makes developing for Cocytus less effort than traditional tool development.

Even more instructive is Cocytus' relation to tgrep2(1). While both tools can do ostensibly the same thing, vertically query and extract subtrees from a set of annotated data, tgrep2(1) is an *order of magnitude* larger than grep(1). Most of this code has to do with supporting tgrep2(1)'s special index format and querying that format. By adhering to a simple format and using brute force parallelization over unused resources, Cocytus can achieve the same speeds with $> 1/10$ of the code complexity. The results are described in Figure 10.

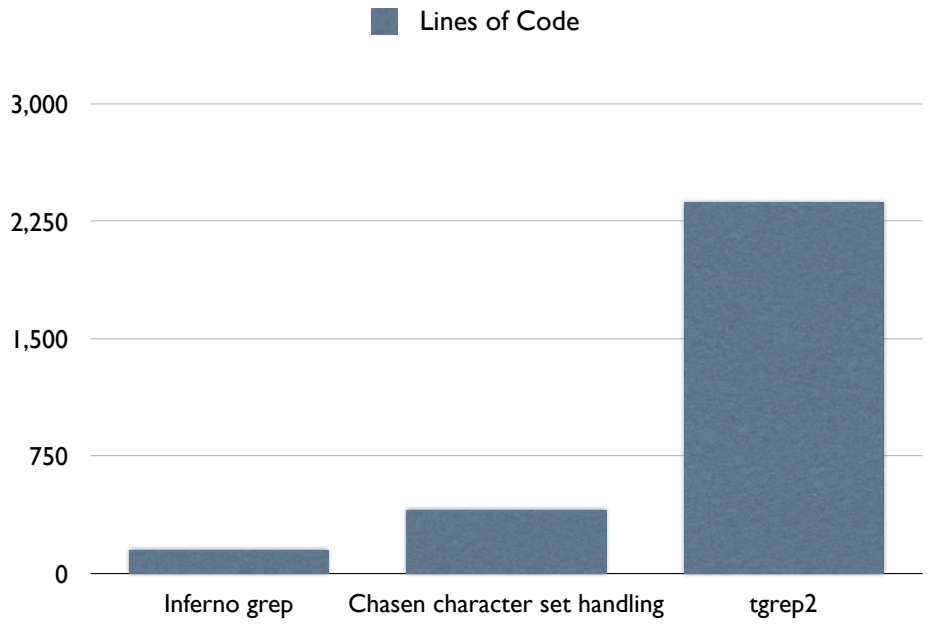

Figure 10. *Amount of code in various tools.*

6. Available at `http://invisible-island.net/c_count/c_count.html`.

7. Discussion

Cocytus takes a completely different approach to developing an NLP platform. Instead of developing an application on top of the system and being forced to rely on the idiosyncracies of that system, we implemented the system on a hosted operating system. That allowed us to avoid many of the problems that plague other systems, like inconsistent resource formats and system behavior.

By putting the responsibility for standardizing resource presentation in the system, we can provide the user with a much more flexible way of interacting with the system. Users are free to create their own patterns from small tools using a persistent interface that is more intuitive and flexible than traditional shells and giving access to commands. In fact, it is not readily obvious to Cocytus users that they are using an NLP platform. The system takes care of the busy work that typically has to be implemented manually.

Free of the implementation overhead that normally hinders the development of NLP tools, institutions can concentrate on developing novel NLP tools and solutions. Previously underutilized systems, lab computers, or researcher workstations can then be repurposed as Cocytus nodes because of the portability and common interface it provides as a hosted operating system.

The key to Cocytus' functionality is namespaces. The ability to represent resources dynamically allows Cocytus to deal with a variety of different resources simply using a common interface. We do not have to develop new protocols or parsers to implement tools on Coctytus. Just by standardizing on two useful abstractions, TreePaths and UTF-8, and transforming namespaces that do not adhere to these standards, we can handle a variety of languages and data formats while relying on a single interface and method of interacting with the system, Unix-style tools. Namespaces also give us distributed computation "for fre" because computational resources themselves are namespaces that can be imported and used by remote systems.

This approach is significantly different from current approaches. Cocytus requires a different mindset and creativity to use effectively. This can be disconcerting for users expecting a traditional system. In essence, the system rewards user mastery at the expense of beginning users, a symptom of many systems. A set of tutorials and screencasts are under development to correct this problem.

As the system improves and matures, we expect it to become more of a foundation for lab's research, enabling researchers to use the system to develop tools that allow NLP work to be done more quickly.

8. Future Work

Cocytus is still in its infancy, so there are many potential paths it can take. We consider the following to be especially promising.

Inferno and its predecessor Plan 9 are both popular operating systems in High Performance Computing (HPC): a version of Cocytus to take advantage of larger HPC systems would be an interesting way of exploring the potentials and limitations of the platform.

Also, automatic character detection can be ambiguous and difficult for small sample sizes. This can be a problem, especially in places where the advantages of Cocytus are most evident, like large unstructured datasets. A way of dealing with this detection problem effectively would help Cocytus in dealing with web datasets.

The current namespace layering is unwieldy. Changing the system to have the transformation system work according to modules similar to Unix's streams IO systems(Ritchie, 1984) might be a better alternative. A new version, with a language for dynamically adding and removing transforms to a namespace, is currently being considered.

The sorting phase of the MapReduce sorts TreePaths but does not sort according to language order. Although natural sorting has many well–known problems (especially for languages where the pronunciation depends on context), combining morphological analysis (extracting the pronunciations) in the Map phase with an intelligent sort could have useful applications in dealing with multilingual language processing.

Currently Cocytus does not have a standard set of NLP tools, like morphological analyzers or syntax parsers. Cocytus relies on calling these commands from the underlying operating system. This defeats the central goal of Cocytus, hiding differences in operating systems and resources. A complete NLP toolchain for Cocytus, supporting a variety of languages, will make it more productive.

To take better advantage of Cocytus' pipelining, we are planning to implement a few tools, including a modular port of awk(1) to Inferno. This will allow the easy implementation of tools that fall between the boundary of the shell and Limbo. The shell is good for small tasks but lacks the ease of floating point handling and the terse syntax of awk for record–based IO. Limbo provides a richer set of tools but lacks the niceties of awk, native support for regular expressions, and a standard method of interacting with data–driven input.

Finally, the added information contained in TreePaths can cause congestion when Cocytus nodes are linked with slow connections. In order to solve this performance problem, an optional module to compress TreePaths using Woods (1983) is planned.

9. References

Bialecki A., Cafarella M., Cutting D., O Malley O., "Hadoop: a framework for running applications on large clusters built of commodity hardware, 2005", *Wiki at http://lucene. apache. org/hadoop*, n.d.

Bird S., Chen Y., Davidson S., Lee H., Zheng Y., "LPath: A path language for linguistic trees", , Unpublished manuscript, 2004.

Church K., "Unix for Poets", *Notes of a course from the European Summer School on Language and Speech Communication, Corpus Based Methods*, 1994.

Cox R., Grosse E., Pike R., Presotto D., Quinlan S., "Security in Plan 9", *Proceedings of the 11th USENIX Security Symposium table of contents*, USENIX Association Berkeley, CA, USA, p. 3-16, 2002.

Cunningham H., Gaizauskas R., Wilks Y., "A General Architecture for Text Engineering (GATE)—a new approach to Language Engineering R&D", *Research Memo. University of Sheffield*, 1995.

Dean J., Ghemawat S., "MapReduce: Simplified Data Processing on Large Clusters", *Communications of the ACM*, vol. 51, n° 01, p. 7, 2008.

Dorward S., Pike R., Presotto D., Ritchie D., Trickey H., Winterbottom P., "The Inferno Operating System", *Bell Labs Technical Journal*, vol. 2, n° 1, p. 5-18, 1997.

Evans N., "Representing disparate resources by layering namespaces", *Second International Workshop for Plan Proceedings*, 2007.

Evans N., Asahara M., Matsumoto Y., "Trees as paths: lessons from file systems and Unix in dealing with language trees", *IPSJ Technical Report*, 2007.

Ferrucci D., Lally A., "UIMA: an architectural approach to unstructured information processing in the corporate research environment", *Natural Language Engineering*, vol. 10, n° 3-4, p. 327-348, 2004.

Hirofuchi T., Kawai E., Fujikawa K., Sunahara H., "USB/IP: A Transparent Device Sharing Technology over IP Network", *Transactions of Information Processing Society of Japan*, vol. 46, p. 349-361, 2005.

Kernighan B., Plauger P., "Software tools", *ACM SIGSOFT Software Engineering Notes*, vol. 1, n° 1, p. 15-20, 1976.

Kudo T., Matsumoto Y., "Japanese dependency analysis using cascaded chunking", *International Conference On Computational Linguistics*, vol. 1, p. 1-7, 2002.

Li S., Momoi K., "A composite approach to language/encoding detection", *Proc. 19th International Unicode Conference*, 2001.

Loper E., Bird S., "NLTK: the Natural Language Toolkit", *Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics*, vol. 1, p. 63-70, 2002.

Matsumoto Y., Kitauchi A., Yamashita T., Hirano Y., Imaichi O., Imamura T., "Japanese morphological analysis system ChaSen manual", *Nara Institute of Science and Technology Technical Report NAIST-IS-TR*, vol. 97007, p. 232-237, 1997.

Pantel P., "Data Catalysis: Facilitating Large-Scale Natural Language Data Processing", n.d.

Pike R., Presotto D., Thompson K., Trickey H., "Plan 9 from Bell Labs", *Computing Systems*, vol. 8, n° 3, p. 221-254, 1995.

Pike R., Ritchie D., "Styx architecture for distributed systems", *Bell Labs Technical Journal*, vol. 4, n° 2, p. 146-152, 1999.

Ritchie D., "A stream input-output system", *AT&T Bell Laboratories Technical Journal*, vol. 63, n° 8, p. 1897-1910, 1984.

Ritchie D., "The Limbo Programming Language", *Inferno 3rd Edition Programmers Manual*, n.d.

Rohde D., "TGrep2 User Manual", 2004.

Thompson K., “UNIX Implementation”, *The Bell System Technical Journal*, vol. 57, n° 6, p. 1931-1946, 1978.

Thompson K., “Hello World”, *Proceedings of the Winter 1993 USENIX Conference*, vol. 1, p. 43-50, 1993.

Winer D., “XML-RPC Specification, 1999”, *URL http://www. xmlrpc. com/spec*, n.d.

Winterbottom P., Pike R., “The design of the Inferno virtual machine”, *Hot Chips*, n.d.

Woods J. A., Finding Files Fast, Technical Report n° UCB/CSD-83-148, EECS Department, University of California, Berkeley, January, 1983.

Ylonen T., Kivinen T., Saarinen M., Rinne T., Lehtinen S., “SSH Protocol Architecture”, 2002.