



11-2009

## Essential Tools: Version Control Systems

Konrad Hinsen

Konstantin Läufer

*Loyola University Chicago*, [klaeufer@gmail.com](mailto:klaeufer@gmail.com)

George K. Thiruvathukal

*Loyola University Chicago*, [gkt@cs.luc.edu](mailto:gkt@cs.luc.edu)

Follow this and additional works at: [https://ecommons.luc.edu/cs\\_facpubs](https://ecommons.luc.edu/cs_facpubs)



Part of the [Computer Sciences Commons](#)

### Recommended Citation

Konrad Hinsen, Konstantin Läufer, George K. Thiruvathukal, "Essential Tools: Version Control Systems," *Computing in Science and Engineering*, vol. 11, no. 6, pp. 84-91, Nov./Dec. 2009, doi:10.1109/MCSE.2009.194

This Article is brought to you for free and open access by the Faculty Publications and Other Works by Department at Loyola eCommons. It has been accepted for inclusion in Computer Science: Faculty Publications and Other Works by an authorized administrator of Loyola eCommons. For more information, please contact [ecommons@luc.edu](mailto:ecommons@luc.edu).



This work is licensed under a [Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 License](#).  
Copyright © 2009 Konrad Hinsen, Konstantin Läufer, George K. Thiruvathukal



## ESSENTIAL TOOLS: VERSION CONTROL SYSTEMS

By Konrad Hinsén, Konstantin Läuffer, and George K. Thiruvathukal

Did you ever wish you'd made a backup copy of a file before changing it? Or before applying a collaborator's modifications? Version control systems make this easier, and do a lot more.

It's a safe bet that everyone reading this article works with files that are regularly modified over a long period of time. Program code is the most obvious example, but scientific publications typically fall into the same category. Those who do system administration can add their computers' various configuration files to the list. And for many of these long-lived and regularly modified files, there's more than one person working on them.

In these and similar situations, something usually goes wrong sooner or later. For example, your program might suddenly stop working correctly or an important paragraph might mysteriously disappear from the paper you're writing. At this point, three key questions arise:

- Which files were changed?
- Who made the change?
- What did the files contain before the change?

A version control system can help you answer these questions rapidly and reliably.

A VCS tracks changes to a set of files—typically a directory's contents—called a *project*. For each change, the VCS records the date and time, the person who made the change, and the differences between the file contents before and after the change. From this information, it

can reconstruct the files' past contents if necessary. VCSs also offer a set of tools to help users employ this information efficiently to solve frequently occurring project management tasks.

### VCS Workflow

To illustrate a typical VCS workflow, we'll use example command lines for Unix computers running the popular distributed VCS Mercurial (<http://mercurial.selenic.com>), but the command lines are similar for other systems. The "Integrated Development Environment Support for VCSs" sidebar shows the same example done inside Eclipse, a popular IDE.

Here, the command `hg` (the chemical symbol for mercury) is the Mercurial program. First, you tell the VCS to create a new directory (called `my_project`) and turn it into a version-controlled project:

```
hg init my_project .
```

Mercurial stores its bookkeeping information in the subdirectory `my_project/.hg`, which is created during initialization. Next, you copy any initial project contents into this directory. If you don't yet have content, you work on the project until you have the first version that you'd like to keep a snapshot of; you then type

```
hg add
hg commit --message "First
version of my project"
```

Mercurial adds these files to the project's list of version-controlled files and records the project's current state as a numbered revision. You need the first command because not all files in the project's directory are automatically version controlled. For example, you wouldn't want computer-generated files, such as compiler output, under version control. To fully control everything that goes into the repository, you can specify the files you want added after `hg add`. By default, Mercurial adds everything.

When committing a revision, Mercurial also records the date and time, and the name and/or email address of the user who committed it. The latter information is typically taken from a configuration file, but you can also specify it on the command line. Finally, Mercurial records the text provided after the `--message` as the commit message. This message is meant for human readers (including your older self in the future), so try to make it informative. If you want to provide more than just a line, you can omit the `--message` option and have Mercurial open a text editor of your choice for typing the message. After committing the project's current state as a revision, you can continue working on your project until you want to commit another revision.

To make this example more concrete, let's say you type the following lines into your computer:

```
hg init my_project
echo "This is my first file"
> first_file
hg add
hg commit --message "First
version of my project"
```

To see what Mercurial has recorded in your project, you type

```
hg log
```

This should yield something like

```
changeset: 0:d6dcac101f82
tag:      tip
user:     Konrad Hinsen
          <hinsenk@cnrs-
          orleans.fr>
date:     Mon Aug 10
          13:47:22 2009
          +0200
summary:   First version of
          my project
```

The first line tells you that this is the information about changeset number 0 (the first—and for now the only—one in your project). A *changeset* is a collection of changes to various files. Revisions and changesets are distinct: a changeset is the difference between two consecutive revisions, and a revision is the result of applying all changesets up to a specific number. Mercurial uses the same number to refer to a revision and to the changeset that immediately leads to the revision. It also assigns a unique identifier to each changeset, consisting of a long number, of which the first 12 hexadecimal digits are also given in the first line. Moreover, a changeset can have any number of tags. *Tags* are just convenient labels for specific changesets

or revisions. Mercurial automatically attributes the tag `tip` to the most recent changeset. You can add other tags at your convenience—for example, `release_2.3`—using the command `hg tag`. The remaining lines show the information that was recorded about the changeset. If you want more details, such as the names of the files that have been changed, you can type `hg log -v`.

To continue with your exploration of how Mercurial works, type

```
echo "This is my second file"
> second_file
hg status
```

This prints

```
? second_file
```

which is a concise status report about your project. The status report lists modified files (preceded by `M`), newly created and not yet added files (preceded by `?`, as shown above), removed files (preceded by `R`), and a few other possible modifications. Here, the question mark tells you that you haven't yet added the file to the version-controlled file set. To do that, you type

```
hg add
hg status
```

The project status now is

```
A second_file
```

indicating that `second_file` has been added. You're now ready to commit your second revision:

```
hg commit --message
"An update"
```

Typing `hg log` will now show two changesets. Let's look at the difference

between the two revisions that correspond to them:

```
hg diff -r 0 -r 1
```

This command produces the same kind of output as the `diff` utility familiar from Unix systems:

```
diff -r d6dcac101f82 -r
09856f1f1133 second_file
---/dev/null Thu Jan 01
00:00:00 1970 +0000
+++ b/second_file Mon Aug 10
14:08:13 2009 +0200
@@ -0,0 +1,1 @@
+This is my second file
```

This instruction set lets you obtain the second version of a file starting from the first version, whose nonexistence is somewhat cryptically indicated by `/dev/null`. Here, the plus sign gives the instruction to "add a line," and the addition's location is given by the specification `-0,0 +1,1`.

With just these few example commands, you can keep track of changes to your files. Mercurial also provides many more commands for more or less common project management tasks, recreating a given revision, exporting and importing changesets for communication with collaborators, publishing revisions on public servers, and updating your local copy from a public server. The most complicated task in working with VCSs, however, is integrating several people's changes into a single, coherent version.

## Resolving Collaboration Conflicts

When more than one person works on a project, conflicts become possible: two or more users might work on the same file and apply incompatible modifications. In practice, the

## INTEGRATED DEVELOPMENT ENVIRONMENT SUPPORT FOR VCSs

When choosing a version control system, one key question is whether it will integrate well with your existing development tools and your preferred workflow. In this sidebar, we'll take a look at integrated development environment support for VCSs, using our favorite IDE, Eclipse, as an example. Other major IDEs also support VCS interaction, but (in our opinion) Eclipse does it best in terms of usability and reliability.

As an extensible development platform, Eclipse can, in principle, support any VCSs. It supports Concurrent Versioning System (CVS) out of the box, and Subversion (SVN) users have two choices of plugins—one, Subversive, is officially under the Eclipse umbrella, but requires that you separately install several components. In all cases, interaction with the VCS is integrated via context menus and node decorations into the standard Eclipse tools (Project Explorer, Navigator, and so on).

For client-server VCSs such as CVS and SVN, Eclipse has separate VCS-specific perspectives (that is, task-oriented organizations of views, menus, and toolbars) for managing repositories and a mostly VCS-independent perspective for synchronizing local projects with server-based repositories. The former, called the *Repository Exploring* perspective (see Figure A), lets us view—and in some cases modify—repository resources without checking out local copies. The latter, called the *Team*

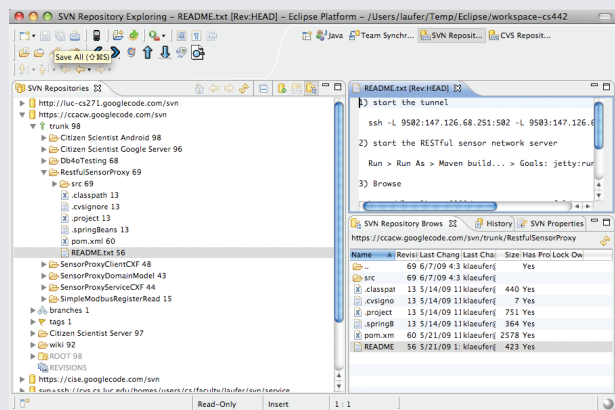


Figure A. The SVN Repository Exploring perspective. These perspectives, which are specific to version control systems (VCSs), let users manage and browse their repositories, as well as view and modify the resources they contain.

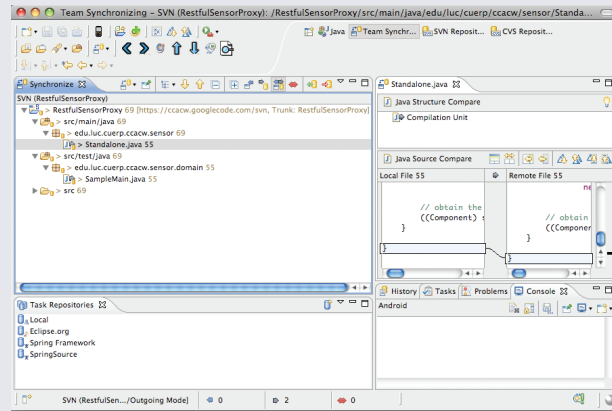


Figure B. The Team Synchronizing perspective. This general perspective lets users interactively control the changes they're about to commit to or download from a repository.

*Synchronizing* perspective (see Figure B), lets us control interactively the exact changes we're about to commit to or download from a repository; in particular, it lets us make structural comparisons between local and remote resources.

As a decentralized, peer-to-peer VCS, Mercurial doesn't require us to manage a list of repositories or synchronize with a server. Accordingly, the Mercurial-Eclipse plugin ([www.vectrace.com/mercurialeclipse/](http://www.vectrace.com/mercurialeclipse/)),

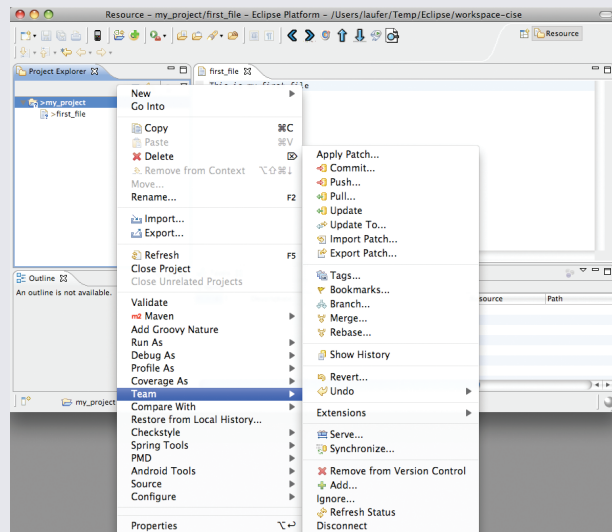
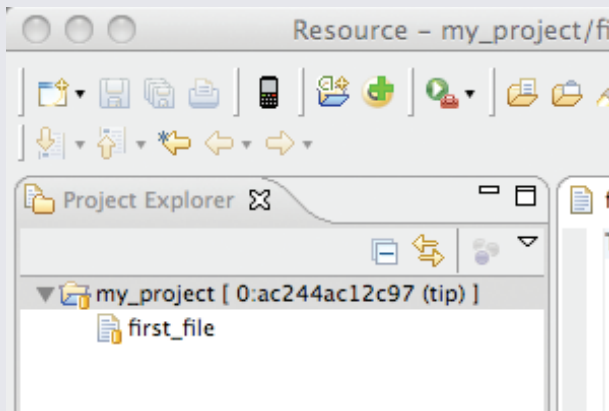
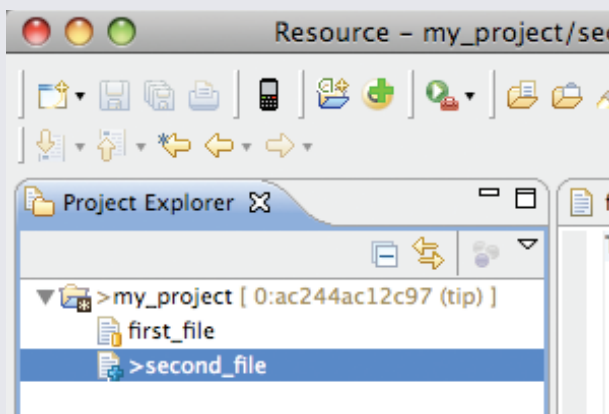


Figure C. Team context menu. This menu exposes most operations available in the specific underlying version control system—in this case, Mercurial.





(1)



(2)

Figure D. Resource status icons. These icons indicate the status of a resource in the repository. (1) The barrel indicates no change, while (2) the asterisk means at least one change, and the plus sign indicates a resource about to be added to the repository.

doesn't include these additional perspectives. After installing the plugin, we simply tell it where to find the Mercurial executable (`hg`), and we're good to go.

We'll now use Eclipse to go through the same example as in the main article. We can easily accomplish the first steps—creating a project and adding a file—through the Project Explorer (see Figure C). Next, to convert the project into a local Mercurial repository, we right click on `my_project` and choose *Team*, then *Share Project*. The icons for the project directory and file now display with a question mark (as with the `hg status` output in the main example) to indicate that they were added recently, but haven't yet been committed. Furthermore, the *Team context* menu now exposes most Mercurial operations, including those we've already seen.

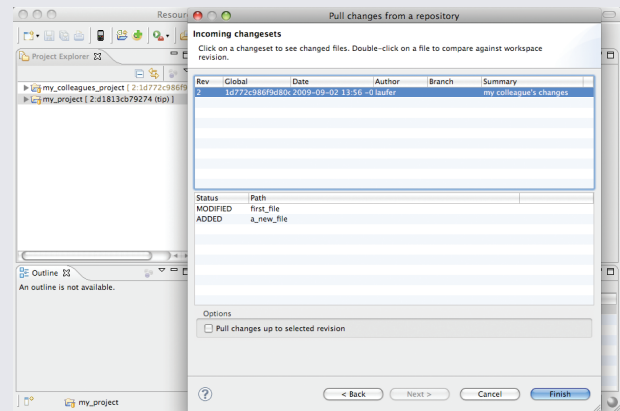


Figure E. Incoming changesets dialog. When we pull changes from another repository, this popup shows us a list of incoming changesets to choose from.

In particular, we can either add and commit specific resources to version control or perform a single commit that includes the desired additions. As Figure D shows, our icons now change to display a barrel-shaped repository symbol, indicating that the resources are under version control, but haven't changed since the most recent commit. This symbol corresponds to a resource not having an entry in `hg status`.

After creating another file and adding it to version control, the project's symbol turns to an asterisk, indicating that there has been at least one change, and the file's symbol turns to a plus sign, indicating that it has been added but not yet committed. Once we commit again, all symbols change back to the repository symbol.

The *Team context* menu doesn't include cloning. Instead, Eclipse supports repository cloning through its *Import context* menu (a *File* submenu). We simply import from Mercurial by choosing the only option, *Clone repository using MercurialEclipse*. We can then specify a remote URL or a local directory from which to import the repository.

The *Team* menu does let us pull changes from another repository by specifying a remote URL or a local directory. We can also inspect the available changesets visually before going ahead with the pull (see Figure E). Once we instigate the pull, a popup displays with the same output as running `hg pull` on the command line.

Finally, we can choose the changeset to merge into the current tip, but there doesn't seem to be a way to choose from among various merge options. We hope that future versions of MercurialEclipse will support these options.

question of whether modifications are compatible is a subtle one, particularly when we expect the computer to decide. However, most VCSs handle typical uncritical situations—such as two users adding a complete function to a source code file at clearly different positions—rather well. But when it comes to real conflicts, don't expect any miracles: the best a VCS can do is warn you about the conflict and ask you to provide the final version of the files that have conflicting modifications.

When a VCS integrates changes made by several people and deals with possible conflicts, it's called *merging*. All VCSs support functions for merging, but they're a little less standardized than the basic operations. Moreover, for all but the most trivial cases, it's wise to use a special tool with a graphical user interface to manually reconcile conflicting changes. You have to separately install and configure such tools. The following example shows how a simple merge operation is performed using Mercurial and the file merge utility provided by Apple's XCode toolkit for the Macintosh. Because Mercurial knows from its configuration file that it should call Apple's merge utility, you don't see explicit references to it in the following.

First, we make a clone, or copy, of the repository `my_project` generated earlier:

```
hg clone my_project
      my_colleagues_project
```

When making a clone, we could use `cp -rp` instead of `hg clone`; the advantage of the latter is that it verifies the repository's integrity and lets us clone repositories from a Web server. In a realistic situation, the clone would be moved to another machine and worked on by someone else.

Next, we add a new file and modify another file in our repository:

```
cd my_project/
echo "This is my third file"
  > third_file
echo "I changed my first
  file" >> first_file
hg add
hg commit -m "some more
  changes"
```

Our colleague also makes some changes to the cloned version:

```
cd ../my_colleagues_project/
echo "A new file" >
  a_new_file
echo "second line of first
  file" >> first_file
hg add
hg commit -m "my colleague's
  changes"
```

Here, we assume that the separately modified repository resides on the same computer or has been copied back there for the merge procedure. (Although Mercurial also has commands for merging over the network or exchanging modifications by email, we won't use them here.) We start the merge operation by obtaining our colleague's changes from his repository:

```
cd my_project/
hg pull ../
      my_colleagues_project/
```

Mercurial provides some information:

```
pulling from ../
  my_colleagues_project/
searching for changes
adding changesets
adding manifests
adding file changes
```

```
added 1 changesets with 2
  changes to 2 files (+1 heads)
(run 'hg heads' to see heads,
  'hg merge' to merge)
```

It's important to realize that up to now, Mercurial hasn't modified any of the project files; it has simply integrated the changes from the other repository into its bookkeeping database. It then indicates that there are two heads. A *head* is the terminal point of a line of sequential changes; it typically represents a project's most recent version. The tag `tip` that we've seen before refers to the head with the highest changeset number. So, because the VCS has integrated a second line of sequential changes, there are now two heads:

```
hg heads

changeset: 3:10874dd8014c
tag:       tip
parent:    1:d15ee4c775ae
user:      Konrad Hinsen
           <hinsencnrs-
           orleans.fr>
date:      Tue Aug 18 12:32:57
           2009 +0200
summary:   my colleague's
           changes

changeset: 2:93406e2bac55
user:      Konrad Hinsen
           <hinsencnrs-
           orleans.fr>
date:      Tue Aug 18 12:31:49
           2009 +0200
summary:   some more changes
```

The next step merges the two heads into one. This is where conflict resolution occurs:

```
hg merge

merging first_file
```

At this point, Mercurial stops and runs an external merge tool. The tool shows the two versions of `first_file` side by side and points to the second lines, which are different. It offers an action menu with five possible choices:

```
Choose left
Choose right
Choose both (left first)
Choose both (right first)
Choose neither
```

We choose both (right first) and then save the file. Mercurial continues:

```
1 files updated, 1 files
merged, 0 files removed,
0 files unresolved
(branch merge, don't forget
to commit)
```

This indicates that the merge went fine and reminds us to commit the most recent changes. Before doing so, let's look at the differences:

```
hg diff

diff -r 93406e2bac55
a_new_file
--- /dev/null Thu Jan 01
00:00:00 1970 +0000
+++ b/a_new_file Tue Aug 18
12:38:50 2009 +0200
@@ -0,0 +1,1 @@
+A new file
diff -r 93406e2bac55
first_file
--- a/first_file Tue Aug 18
12:31:49 2009 +0200
+++ b/first_file Tue Aug 18
12:38:50 2009 +0200
@@ -1,2 +1,3 @@
This is my first file
+second line of first file
I changed my first file
```

The final step is the commit:

```
hg commit -m "Merged with my
colleague's changes"
```

No action is required for the new files `third_file` and `a_new_file` that were created independently as there's no source of conflict and the final repository contains them both.

Although the merge process might seem complex, imagine what you'd do without a VCS. You'd probably apply a tool like the Unix command `diff` recursively to the whole project, and examine all the changes on your screen to spot possible conflicts. There's a good chance you'd miss one, which is indeed a frequent source of subtle mistakes in collaborative projects. A VCS helps you reconcile conflicting changes. Moreover, it keeps a detailed trace of everything that happened, so that any project member can verify at any time whether the merge was done correctly.

## Centralized and Distributed Systems

The first free, open source project-oriented VCS was the content versioning system (CVS) published in 1990. CVS uses a client-server architecture in which all project data is stored on a server. Every project collaborator has a client software on his or her computer that connects to the server through a network. Thus, a server administrator must set up the project. The administrator defines each user's access rights and manages backups and other maintenance operations. CVS became very popular in the open source world and was the basis of the first collaborative servers such as SourceForge.

Today, CVS has been replaced almost completely by Subversion, or *SVN* (the name of the command-line tool that implements the client protocol).

SVN is basically an improved implementation of CVS ideas; it maintains CVS's server-client architecture and provides an almost identical command set. SVN's main innovation is the notion of transactional commit semantics, which is similar to the concept found in relational database systems. Transactional semantics ensure that a commit is either performed completely or not at all. This approach thus prevents the repository from being in an inconsistent state if a network problem interrupts a commit. SVN also adds directory versioning (including file renaming), constant-time branching and tagging, and space-efficient differences between binary files. Currently, SVN is by far the most widely used VCS in the open source world and enjoys significant popularity in commercial environments as well.

The main problem with centralized VCSs such as CVS and SVN is that they depend on a server and a network connection. The server stores the only master copy of the whole project. If the server becomes unavailable, nobody can work on the project. If the server's data is lost, the project is lost as well. Moreover, work on the project is possible only with a network connection. Because many software developers like to work offline (to avoid Internet distractions) or while traveling, they often make commits when a network connection is available rather than when the project's state justifies them.

Distributed VCSs address this problem. With a distributed system, there's no server. Each user has a full copy of the whole project—in the form of a directory—on his or her computer. The distributed VCS attaches bookkeeping information to each directory for its own use. The earlier Mercurial example illustrates how a distributed VCS works. If only one person is

working on a project, the set up is obviously simple. However, most projects have several collaborators, and that's where distributed VCSs can get a bit complicated to use. In fact, project collaborators must agree on a strategy for sharing modifications and synchronizing their local project copies. One such strategy is to adopt a central master server (as for a centralized system).

Although distributed VCSs have been around for a while, they've only recently become popular with the advent of several second-generation systems that now compete for developers' attention: Bazaar, Darcs, Git, and Mercurial. Each has been adopted by a few big and well-known projects, and each has its advocates who claim it's the best. In practice, each will work fine for most projects; differences emerge only in extreme situations. Although distributed systems aren't yet threatening SVN's market dominance, more and more open source projects—including well-known heavyweights such as Linux, Mozilla, and Python—are switching to distributed version control.


Very recently, “super clients” have emerged to give users the best of the centralized and distributed worlds (<http://blog.red-bean.com/sussman/?p=116>). The basic idea of a super client—such as hgsubversion, which permits access to Subversion repositories from Mercurial—is to clone an existing server-based repository into a local (distributed) repository. This can be especially helpful when your project takes an experimental direction and you want to track your changes locally, without formally committing them on the official central repository. You can later work with the official repository's maintainers to push your change-sets back upstream. Alternatively, you can create a new project altogether. Distributed VCS technology, therefore,

has the potential to be greatly democratic and liberating or wildly anarchical (much like the Wikipedia model sans the recent editorial oversight provisions). While it's beyond our scope here to discuss software project management, it's clear that some combination of centralization and distribution is the right mix for most real-world projects. That said, most computational science projects are experimental and exploratory in nature, and often take existing code and evolve it for new needs. Given that, we certainly like what we see in distributed VCSs.

So, which VCS is right for you? Obviously, there's no single answer that works for everyone. All of the widely used systems work well, so you can't make a serious mistake choosing one or another. If you join an existing project and want your changes to be recognized by the project maintainers, you have no choice but to use their system. If you want to use a collaborative development site, your choices are limited as well: SourceForge proposes SVN, Git, and Mercurial, while GoogleCode has SVN and Mercurial, and so on. For a new project, the only important decision is between a centralized system (probably SVN) and a distributed one. As a rule of thumb, pick a distributed system unless you have a good reason not to. When choosing among the four big distributed systems, consider practical criteria:

- Do you know experienced users who can help you?
- Can you get easy-to-install distributions for all of your computers?
- Does the VCS integrate well with your existing development tools and your preferred workflow?

- Does the documentation look understandable to you?

Finally, in the unlikely case that you choose system A and run into serious limitations a few years later, you can always switch to system B and convert your existing repositories; various tools exist to help you with such a migration. 

---

**Konrad Hinsén** is a researcher at the Centre de Biophysique Moléculaire in Orléans (France) and at the Synchrotron Soleil in Saint Aubin (France). His research interests include protein structure and dynamics and scientific computing. Hinsén has a PhD in theoretical physics from RWTH Aachen University (Germany). Contact him at [konrad.hinsen@cnrs-orleans.fr](mailto:konrad.hinsen@cnrs-orleans.fr).


---

**Konstantin Läufer** is a professor of computer science at Loyola University Chicago. His research interests include programming languages, software architecture and frameworks, distributed systems, mobile and embedded computing, human-computer interaction, and educational technology. Läufer has a PhD in computer science from the Courant Institute at New York University. Contact him via [www.cs.luc.edu/lauder](http://www.cs.luc.edu/lauder).

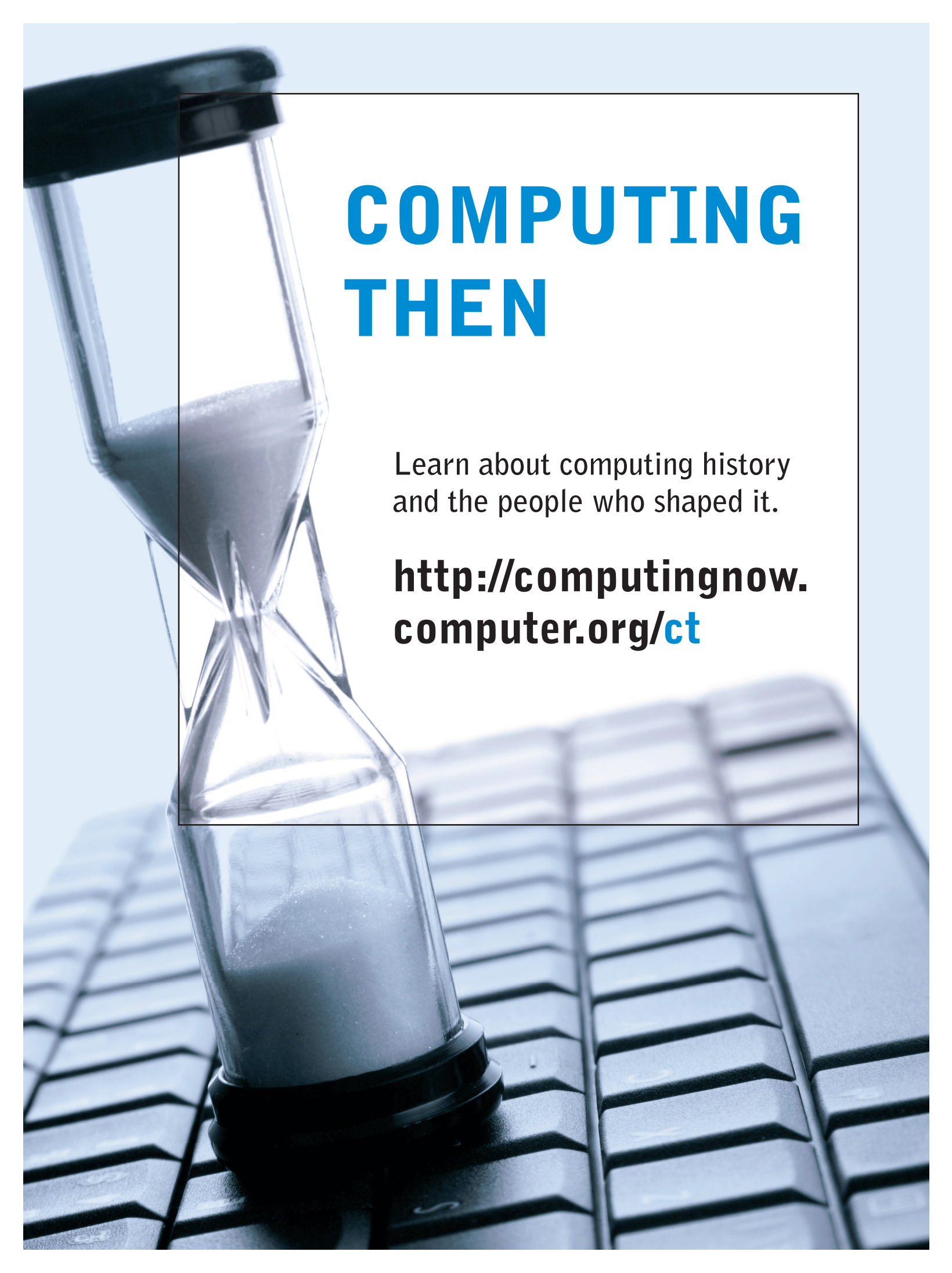
---

**George K. Thiruvathukal** is an associate professor of computer science at Loyola University Chicago and associate editor in chief of *CiSE*. His technical interests include parallel/distributed systems, programming language design/implementation, and computer science across the disciplines. Thiruvathukal has a PhD in computer science from the Illinois Institute of Technology. Contact him via <http://gkt.etl.luc.edu>.

---

 *Selected articles and columns from IEEE Computer Society publications are also available for free at <http://ComputingNow.computer.org>.*



An hourglass with sand falling from the top bulb into the bottom bulb, resting on a computer keyboard. The hourglass is tilted, and the sand is in motion. The keyboard is black and has several keys visible. The background is a light blue gradient.

# COMPUTING THEN

Learn about computing history  
and the people who shaped it.

**[http://computingnow.  
computer.org/ct](http://computingnow.computer.org/ct)**