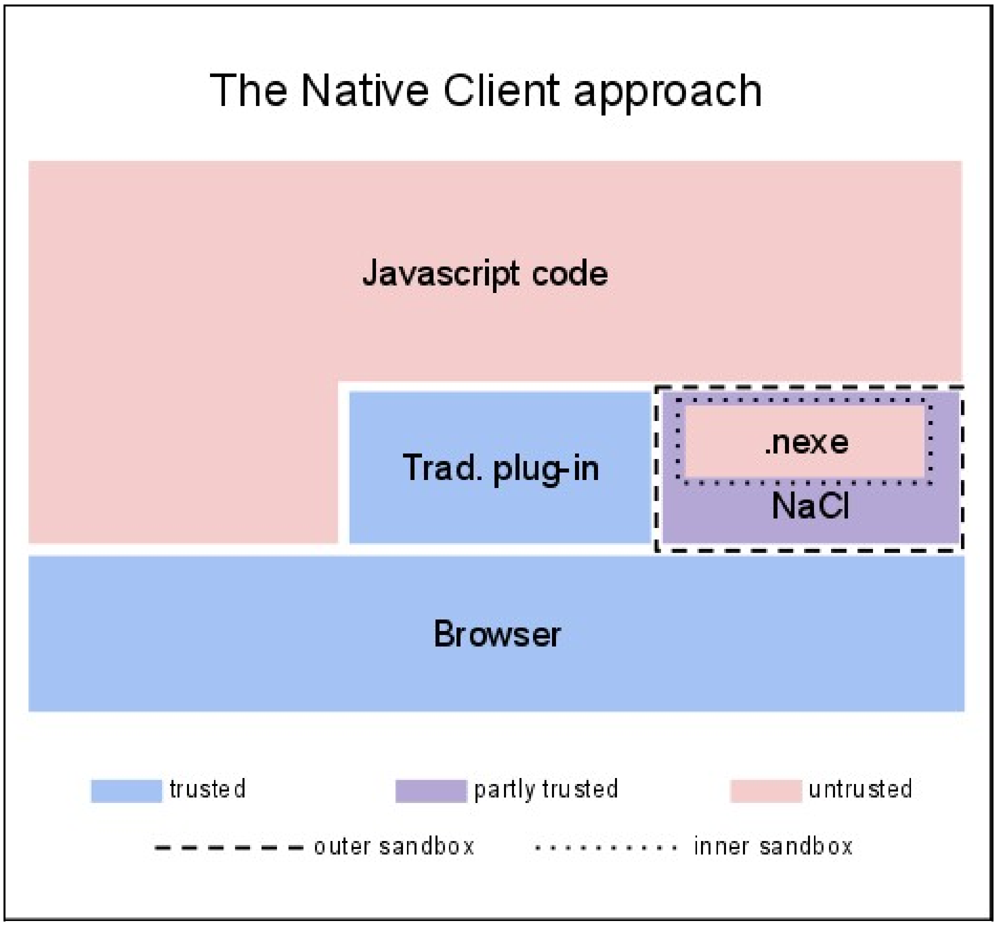
NaTcl : Native Client Tcl Port

NaTcl : Native Client Tcl Port

Alexandre Ferrieux, France Telecom

1. What is Native Client ?

In 2010, Google started the NativeClient, aka "NaCl" project, which is a new sandboxing paradigm for browser expansion. The idea is to get the best of two worlds: the speed of (nearly) native code, and the safety of sandboxed environments. The various trust boundaries are illustrated below:



This little miracle is achieved by jailing the native code (".nexe") inside two sandboxing layers:

- the outer sandbox:

  This is a traditional process-level sandboxing (chroot, ulimit, etc). It encloses the entire NaCl plugin-process.
- the inner sandbox:

  This one is the real jewel inside NaCl. It is a machine-code-level verification pass and execution context that is run when loading the .nexe, applying an extensive list of checks, among which:
    - ♦ no dangerous instructions (like the one invoking syscalls)
    - ♦ all constant jumps fall on N-byte boundaries and in allowed range

    - ♦ all computed jumps are preceded by an AND operation restricting them to N-byte boundaries and allowed range
    - ♦ the runtime narrows the addressable memory (x86 segment registers)

  These constraints together make it impossible for malevolent code to hide syscalls, either in shifts of the instruction decoding frame, or in data (which are necessarily not executable).

  For code to be eligible as an .nexe, it must be compiled and linked with a modified gnu toolchain guaranteeing the invariants above. Any violation implies instant rejection at load time.

  The N-byte boundary scheme does impact both code size (padding by NOPs) and speed (L1i cache). The NaCl team says they are moderate though. Our own Tcl case demonstrates that the performance loss (wrt truly native Tcl) is indeed bearable.

This double safety legitimates Google's boasting a bulletproof plugin architecture; moreover, the complete isolation from the OS implies that .nexes are only processor-specific: all exchanges with the outside world (the Chrome browser) are done through a new API (aptly named "Pepper" in this salty context). So an x86 .nexe will run unmodified on Windows, Linux, and x86-MacOS.

The Pepper API, which is still in fast expansion, progressively opens up various goodies to the .nexe:

- exchanges with the Javascript context
- sound
- direct access to the frame buffer
- (soon) access to accelerated 2D and 3D graphics

However, some things will by definition never be allowed within NaCl:

- naked sockets
- access to the whole local filesystem

This is obviously the price to pay for the absolute trust that NaCl aims to deserve.

2. NaTcl

2.1 Why ?

In April 2011, Google opened NaCl to outside developers. The motivation for porting Tcl to it stemmed from a general frustration about not (easily) having Tcl in browsers. To the non-JS world, NaCl comes across as an opening to alternate languages.

(for other -- and promising ! -- methods to bring Tcl into the browser landscape, see Steve Landers' paper.)

But the real trigger was Colin Mc Cormack's unwinking enthusiasm and support, backed by his deep knowledge of the whole field (WubTk in perspective).

2.2 Wait a minute

The salient issue that comes to mind when thinking about an NaCl port of Tcl is clearly the isolation from the OS. First, one may ask, How are we supposed to do interesting things in such a neutered environment ?

The answer is, of course: use the browser (and its JS context) as a proxy to the real world. Despite the limitations mentioned above, it can still do many things: GUI (of course); fetch intra-domain URLs; access app-restricted local config or user-selected normal files.

Bottom line: we don't need those missing syscalls anyway !

2.3 How ?

Given the unavailibility of syscalls at link level, two approaches were considered:

- cut "high" : separate Tcl's language and data manipulation core from more peripheral OS-related primitives;
- cut "low" : take it as a whole, faking syscalls.

While the first approach is cleaner, it implies a fair amount of code surgery, which in turn makes it hard to keep in sync with the mainstream codebase. Cutting "low", on the other hand, means a very small set of changes, at the expense of error message clarity ('no such file or directory' instead of 'invalid command name "open"').

After a couple of nanoseconds weighing the options, cutting low sounded like the way to go. This means that the starting point of the porting effort is a list of trivial syscall/libc definitions, typically setting errno to something not-too-alien, and returning the adequate value for failure (NULL or -1). See naclMissing.c.

Once syscall plugging was done, a few ancillary adaptations followed:

- Compatibility headers defining the (unused) structs passed to the emulated syscalls, not provided by the NaCl toolchain's includes. See naclCompat.h
- Toplevel bootstrapping glue calling Tcl_CreateInterp(), wrapping init.tcl, and passing data back and forth to JS. See naclMain.c
- JS support code. See loader.js.
- Incremental build system adaptations: parameterize and call ../unix/configure; patch the generated Makefile.

The bootstrapping code circumvents the absence of local filesystem access by stringifying the contents of init.tcl. This was preferred over a full-fledged VFS by the same reasoning as above: to keep it incremental, refrain from pulling in a significant mass of code.

Note that init.tcl is the only file needing this special handling, because once the interp is initialized, Tcl scripts can take over. For example, [source] is emulated (in init.tcl) by a Tcl coroutine that yields back to JS while the requested URL is fetched by the browser (with a vanilla XHR).

A further motivation for this approach is size and modularity: .nexes tend to be hefty, so as soon as at least two NaTcl-based applications exist (wishful thinking), it is best to share the generic Tcl .nexe in the browser's cache and let the individual apps [source] their specific code (which may be cached too) at init time.

2.4 Putting the pieces together

Once we have a working Tcl interpreter, properly adapted to the peculiar syscall-less link environment, the next step is to integrate it into the JS context's lifecycle. This task is outstandingly easy when Everything Is A String ;-). To be fair, JS also takes part in this, with its own eval() function. Indeed, we can set up a very simple "JS trampoline":

-
  ```
  (JS) String result = natcl.eval("some Tcl code");
  (JS) eval(result);
  ```

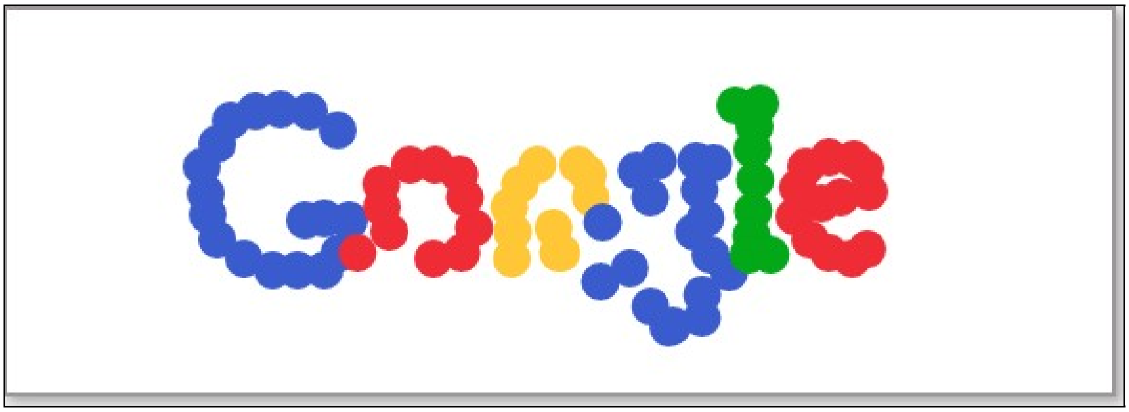
It is important to note that these two lines are not in a tight "while(true)" loop; instead, they are typically invoked from within a JS event handler, which in turn may be set up by (a side effect of) the "eval(result)" line. As a consequence, as long as "some Tcl code" takes a small time to complete (or to [yield]), the JS interpreter and associated browser-borne GUI stay responsive. The coupling between NaTcl and the browser is thus identical to the Tcl/Tk one.

2.5 First real example: the "balls" demo

One of the many showcases of HTML5 features is the Google "balls" demo at

http://www.html5canvastutorials.com/labs/html5-canvas-google-bouncing-balls

It is a modest JS script simulating bouncing balls relaxing to fixed positions drawing a Google logo, and disturbed by the hovering mouse:



It is an interesting porting exercise for NaTcl, because:

- it features quickly-moving graphics (at 30fps)
- it also involves a bit of physics calculations
- it leverages the browser's beautiful antialiased circles

An additional self-imposed constraint was to use a Tk-like API in the NaTcl script. This is at variance with the natural JS canvas API, which is lower-level (exposes a Repaint callback and immediate-mode graphics). But as it turns out, bridging this gap is fairly simple. Basically, it amounts to mapping the current state (items, coordinates) of the Tcl-level canvas to a JS data structure used in the JS Repaint function.

This setup allows the interactive loop to only exchange with Tcl a (stringified) array of integers, feeding them into a Repaint function that was typically JIT-compiled once for all. The resulting speed is adequate, in that 30fps can still be held on an average-powered laptop.

2.6 Performance analysis

(to be completed with current Nacl+Chrome)

Bottom line:

- the NaTcl balls demo uses roughly thrice the CPU used by the original pure-Javascript code at the same frame rate.
- pure Tcl code, not hampered by the I/O with the JS context, runs marginally slower than native Tcl on the same platform.

One thing about the string I/O bottleneck: the NaCl team promised the advent of TypedArrays in the Pepper API, which will allow to populate JS data with native values (like lists and integers) from within NaCl. This points to a promising optimization of the transmission of a bunch of coordinates, directly from Tcl's Lists and Integers to JS's. TBC, when Google delivers.

2.7 NaTk

The "balls" demo shows that, with NaTcl in hand, a JS newbie (like me) can whip up a non-ridiculous coupling with the HTML5 canvas. The fundamental reason is that while the String is a handy common ground, each side knows to back it with more efficient representations.

Now, within this general string-coupling strategy, many forms of Tcl-side syntax and JS-side tricks are obviously possible. In particular, if you replace the JS newbie with a JS+Tcl expert like Colin, you get NaTk (based on ideas from WubTk). Learn more about it in Steve's paper.

3. Ecosystem

Despite the OS agnosticism, the portability dream is a bit spoilt by having NaCl only on Chrome (or Chromium) right now. Though the project is opensource, and Google initially targeted it as a multi-browser plugin, the reaction from competing browsers has been, as could be expected, lukewarm to say the least. Tough.

Still, NaCl retains some headroom in two areas:

- The Chrome App Store: there, dependency on Chrome is by design. Moreover, the download size is also part of the tradition, since the apps are installed locally (in a more persistent form of cache). Find a killer app, write it in NaTcl, publish, reach fame, then don't forget to mention "Powered by Tcl" ;-)
- The Android browser. The NaCl inner sandbox also exists for ARM CPUs (though in a less polished state than x86 and x86_64), and the NaCl team is committed to integrating it into the Android browser as soon as the x86 branches' bugcount reaches zero.

4. Afterword

When we were all mulling over Tcl and browsers in the Spring 2011, various ideas were discussed, among which Steve's amazing ones. In hindsight, NaTcl is less sexy than them, especially with its position under fire in the browser war. Still, it strikes a different balance between effort (minimal) and outcome (medium). And anyway, the observation of intimate contact between Tcl and Javascript was personally enriching.

ACKs

- Colin McCormack, for the initial spark, many good ideas and optimizations, and NaTk.
- Cameron Laird, for patient proofreading and key side-questions
- Brad Chen (from Google), for his sheer skills at taming rogue instructions on any processor
- Steve Landers, for exploring the opposite approach and succeeding !

Bibliography

- NaCl page on Google Code: https://sites.google.com/a/chromium.org/dev/nativeclient
- NaCl inner sandbox concepts by Brad Chen: http://www.youtube.com/watch?v=L8m9U7p_Ntk&feature=related
- NaTcl branch on core.tcl.tk: http://core.tcl.tk/tcl/timeline?r=ferrieux-nacl
- Wiki page by Colin et al, to get started with NaTcl: http://wiki.tcl.tk/28211