Extremely Portable Dyalog APL

Dyalog'12 Conference Version

Objectives

We want to be able to rapidly port Dyalog APL to a many new platforms. At the same time, we want to provide "rich" development environments (similar in functionality to the current Windows IDE) on as many platforms as possible - not necessarily exactly the same sets of platforms as APL will run on. On the platforms where APL itself runs, we want to provide the APL developer with access to local libraries and frameworks for producing graphical applications, TCP connectivity or server-side components, depending on the type of application.

Engines and RIDEs

We believe that the objective will become easier to achieve if we split the current product into three components:

- An "APL Engine" which only makes use of a handful of operating system features (essentially file and TCP socket I/O).
- A "Remote IDE" which makes use of GUI features on a wide variety of platforms to provide access to APL Engines running locally or any machine which can be reached over the network
- A set of "Bridges" to object-oriented platform frameworks: The Microsoft.NET bridge has existed for more than a decade. An experimental Java bridge exists. Suitable platforms need to be selected for each target platform (where available).
- (Can't count to 3): Dyalog's TCP library Conga is already a separate component; wherever possible an APL Engine will need to be accompanied by a version of Conga, to make it useful for building (securely) communicating applications.

The Engine and the RIDE will communicate with each other using a standardized protocol (current prototypes use XML but there may be better alternatives, for example JSON). Dyalog will publish the protocol (after it stabilizes), allowing 3rd parties to build RIDE components (or indeed alternative language engines) targeting environments where Dyalog has yet to tread. For example, it would be nice to invite and/or partly fund students to build an Eclipse front-end using the RIDE protocol.

Process Manager / Proxy

It is expected that RIDE and Engine will most often use (optionally secure) TCP sockets to communicate with each other, although we will layer the architecture to allow other forms of communication in the future. However, we are likely to use sockets even when RIDE+Engine are hosted in the same executable[1], in order to avoid implementing our own asynchronous messaging mechanism.

In scenarios where the RIDEs and Engines are on separate machines, and on multi-user machines, security constraints will usually dictate the use of a "process manager" or Proxy. The Proxy will be responsible for validating credentials presented by a RIDE and determining which pre-existing Engine processes it is

[1] Likely on IOS, Android and other smartphone/tablet environments, and possibly also on the desktop.

allowed to connect to. If the security configuration allows it, the Proxy will also be responsible for starting new Engine processes on request by RIDEs (and potentially also under program control from applications like “Parallel Each”, which need to start remote processes).

Depending on the security / firewall environment that we need to operate in, we will probably need to be able to configure the system to support a variety of scenarios, including:

- Engine listens for a direct connection from a RIDE (with restrictions on where the RIDE can be running)
- RIDE listens for direct connection from an Engine which it has [caused to be] launched
- Engine connects to Proxy after startup; RIDE connects to Proxy which acts as relay for all messages (for scenarios where firewalls limit the number of ports that can be opened)
- Engine registers with Proxy on startup; RIDE initially contacts Proxy but Proxy instructs Engine to open a port briefly in order to allow RIDE to make a direct connection.

The first two scenarios will be relevant when the RIDE and the Engine are running in a “local” environment where security is not an issue. Secure multi-user use will generally require a Proxy.

Target Environments

The following target environments are under consideration:

Target	UI Component for RIDE	Bridges for Engine
Windows Desktop	WPF	.NET, Java
Windows RT (ARM & x86)	Metro	WinRT
Linux (ARM & x86)	Java	Java
Android	Java	Java
Apple iOS	Cocoa	Java, Cocoa
Apple OSX	Java or Cocoa	Java, Cocoa
AIX	N/A	Java
Linux/z	N/A	Java

First Wave

We should target having “beta” versions of a “first wave” of environments within 12 months following the release of 13.2. Whether the support for new platforms will gradually become released based on 13.2, or aim for version 14.0, is (far) too early to say.

RIDE Platforms: Under Windows, since much code can be shared (and reused from the Silverlight project), we should target a WPF-based RIDE for Windows desktops and a WinRT-based RIDE for “Windows 8 UI” at the same time. The next priority is a Java RIDE which will be able to run on most platforms and be used as a stop-gap on platforms where we do not yet have a native implementation.

Bridges: A Java bridge will allow some form of GUI development on a wide variety of platforms, and is therefore the top priority in terms of bridges.

New Engines: In addition to developing a “stand-alone Engine” for existing platforms (Windows, Linux, AIX) target Apple OS/X and Android.

The first Apple OS/X version may need to make do with a Java RIDE and a Java Bridge, until we have time to implement a native RIDE and a Cocoa Bridge.

ARM Linux (Raspberry Pi) and Apple iOS may be allowed to jump the queue due to marketing importance and unstoppable developer enthusiasm on "Dyalog Fridays". Note that no UI component is listed for AIX and Linux/z, it is assumed that a RIDE will be used under Windows or Linux (or on an iPad, Android or Windows8 tablet, etc).

RIDE v1.0 Functionality

It is expected to take at least 2-3 releases before RIDE can be expected to compete with the existing Windows IDE. However, RIDE is expected to be competitive with the TTY interface from day 1. The functionality of RIDE v1.0 needs to include:

1) Enter APL expressions and receive output
2) Handle quote and quote-quad prompting/input
3) Select font and font size for the session and editor
4) Autocomplete input
5) Edit code (functions, classes, namespaces) – and ideally also simple variables
6) Trace /single-step execution (same feature set as Windows IDE)
7) Provide syntax colouring in the session and in edited code
8) Display a Language Bar with language hints
9) View the list of active threads, switch threads, and perform the operations available from the Threads menu under Windows (some early adopters of RIDE will want to debug server applications)
10) Send weak and strong interrupts
11) A few APL hooks for applications: Set title for session tab, display SVG (or other relevant) RainPro output into a window displayed by the RIDE.

The "obvious" next step for subsequent RIDE releases include:

- The functionality provided by the "Windows Explorer" and search tools.
- The ability to locally print or save PDF or other documents produced by the server application.

Ideas / Notes and Technical Requirements

- The protocol must be suitably versioned, so that back-dated components will be able to communicate with each other as the protocol evolves. Once an organization starts using RIDEs+Engines, wholesale replacement of all components at once is likely to be a near impossibility.