NCN/A3R Native Application Framework

("Native-Cloud/Native" services and "Application-as-a-Resource")

NCN/A3R (hereafter **NA3**) is a **QT**-based application-development framework which prioritizes hybrid solutions combining cloud and desktop/native components. The **NCN** (Native-Cloud/Native) model refers to desktop client applications that are integrated with Cloud/Native back-ends; by sharing code libraries and data formats across both end-points, **NCN** solutions are more streamlined than native front-ends with generic back-ends, or Cloud/Native back-ends with web-application clients. The **A3R** (Application-as-a-Resource) model promotes self-contained, downloadable applications that can be distributed in source-code fashion and compiled with few (if any) non-**QT** dependencies. The combined **NA3** framework yields a comprehensive application-development toolkit with numerous components to streamline the implementation of **QT** applications (**NA3** can also be used as a template for implementations based on frameworks other than **QT**, such as wxWidgets or Operating-System-specific options).

The Current Status of QT Cloud Integration

There has been considerable demand in the native-application sector for a systematized Cloud Services model designed to interoperate with cross-platform native applications. Cloud/Native components can augment the functionality of native/desktop software by providing remote storage for user data; enabling users to share content for collaborative work; maintaining domain-specific repositories (i.e., spaces of resources whose format is specialized so that only select applications can access them properly); and upgrading or extending applications without re-install. We use the term "Native-Cloud/Native" to describe hybrid applications whose server and client endpoints are both internally native – in contrast to conventional Cloud/Native where native servers are paired with (potentially) non-native clients.

Cloud/Native support in existing cross-platform native-application frameworks is fairly primitive. Since **QT** is by far the most widely-used such framework, the **QT** case is instructive. In 2013 (following an earlier beta phase) the **QT** company introduced "**QT** Cloud Services", which provided a convenient, **QT**-aware cloud-hosting platform for **QT** accounts (in the company's words: "Qt Cloud Beta has solved an immense need for **QT** developers when it comes to backend-as-a-service and believe that there is an even greater need to provide the **QT** ecosystem with an all-in-one **QT** solution for cloud computing"). However — to the consternation of the **QT** community — this project was discontinued several years later (retroactively we can identify some design flaws which might have hindered the project). Meanwhile, OpenShift discontinued their free-tier Cloud/Native hosting last year, and another company with Cloud/Native options, Arukas, is folding at the end of this month. This means that **QT** developers have limited options even for hosting hand-rolled **QT** cloud solutions (which can be done by compiling **QT** into a Ubuntu container)

Considering the prominence of both **QT** and Cloud/Native technologies in the contemporary computing landscape, it is disconcerting that no standard framework or hosting service provides a cloud platform which works with **QT** "out-of-the-box." The existence of such a platform would be

a boon to software in sectors like scientific computing, bioinformatics, bioimaging, pharmaceuticals, academic publishing, and other fields where (due to complex **GUI** and/or data-analytic requirements) the software is predominantly native-compiled and desktop-oriented.

Of course, many desktop applications have some web integration, but the current architecture forces the client-facing and web-facing components of the application to be almost completely separate, which adds to development time and expense. Moreover, current native-application environments do not fully leverage Cloud/Native services; they may well be implemented via more old-fashioned non-cloud servers. The great possibility of a "Native-Cloud/Native" approach is a peer-to-peer client-server relationship, sharing libraries and data formats on both ends; and the infrastructure to bring the benefits of Cloud Computing (e.g. faster development and deployment, and less expensive hosting, as compared to non-cloud web services) to the native-application ecosystem.

Native-Cloud/Native in the Context of NA3

In light of the Cloud/Native limitations just identified, LTS intends to contribute tools or hosting arrangements that would bring some of the capabilities of **Qt** Cloud Services back to the market. The simplest commercial model for such a product is to license a containerized **Qt**-based **HTTP** server that can run as a local application during testing and development, before being deployed to a container hosting service. We have implemented a prototype server along these lines that we call NDP Cloud (for "Native-Driven Platform on the Cloud"). NDP Cloud is fully self-contained in a **Qt** context (it bundles portions of the Node.js code base and utilizes the **Qt** network module, so it requires no external **HTTP** or sockets libraries). One significant benefit of NDP Cloud is that it is fully transparent: all of the code for parsing and routing **HTTP** requests can be loaded into **IDE**s (such as **Qt** creator) and examined by the debugger. Another benefit is that project-specific libraries can be compiled into both NDP Cloud instances and client front-ends; therefore, clients and servers can share procedures for serializing and deserializing domain-specific data structures. For development and prototyping, NDP Cloud can be launched as an ordinary (non-virtual) Operating System process, which can receive requests via **HTTP** but also via sockets, or the command line (allowing request-management logic to be tested in abstraction from the **HTTP** layer). Further testing can then be performed running NDP Cloud as a local Docker container, before eventually deploying the application to a remote Docker hosting environment.

Via NDP Cloud, **NCN** applications can be deployed on any Docker cloud service, such as OpenShift. In this guise LTS has no direct involvement with the hosting service (although NDP Cloud includes some tools to streamline cloud deployment). Ideally, however, LTS would like to secure its own hosting capabilities, perhaps by using an LTS-specific container deployed on OpenShift or a similar platform. LTS would allocate cloud assets to NDP Cloud licensees (e.g. a limited free-tier hosting plan) for testing and development. A further possibility is to provide free hosting, subject to data-space constraints, to scientific institutions. The dwindling availability of free-tier Cloud/Native options is a hindrance to projects' adoption of Cloud/Native solutions for sharing and disseminating scientific data; this can result in researchers hosting data sets on platforms such as Mendeley or DataVerse, which have limited functionality or customizability compared to Cloud/Native containers. Use-cases for **NA3** in the context of scientific data sets are explained in the discussion of **A3R** below.

Application Development via A3R

The A3R model facilitates implementation of standalone native applications, whose data models and UI logistics are described via integrated metadata. As much as possible, A3R applications are entirely self-contained, so that all application code and data can be packaged into single downloadable resource. In a QT environment, QT modules are available for concerns such as networking, database management, C++ reflection, XML or JSON parsing/querying, and embedded web viewers, so that A3R can leverage these capabilities without requiring separate library installs. For many use-cases, then, an entire desktop application can be deployed in source-code form, to be compiled and launched via a single click within QT creator. Self-contained in this manner, A3R applications can be treated as single resource units — somewhat analogous to container images, but achieving their autonomy by leveraging the QT ecosystem rather than by virtualization.

A3R applications are also autonomous resources by virtue of detailed metadata bundled with application code. This metadata provides a summary of application-specific data models, capabilities, UI features, and user documentation. The metadata may be accessed by human users or by automated tools to help users become familiar with a newly-acquired A3R resource.

The A3R architecture is especially warranted when applications are designed to work with one or several non-standardized, domain-specific data formats (including those unique to an individual data set). In these scenarios, A3R applications provide both libraries for parsing and manipulating the domain-specific formats and a "reference implementation" documenting the proper visualization and User Experience optimal for the unique data structures involved. This structural profiling is advanced not only by domain-specific GUI components implemented within A3R applications, but also by A3R metadata which describes application-specific data types, and declares interface requirements for any software components targeting such data types.

The A3R toolkit includes numerous components which may be useful to programmers implementing cross-platform, desktop-style applications, including a library for in-memory hypergraph-structured data, a built-in database engine for persistent data, a parsing and grammar library, and a foundation for building customized scripting languages. These components have no external requirements and are distributed as raw C++ files that could be dropped in to any QT/C++ project. As with NDP CLOUD on the server side, these components are therefore "transparent": their code is directly bundled with application sources, and may be clearly examined in a debugging session. This is in contrast to typical libraries providing application-development features such as database engines or code parsers, which typically require separate installation (often with separate build tools) and are opaque to the debugger. Another benefit of using internal A3R components is that they can be simultaneously compiled into NCN instances developed alongside them. With that said, developers could certainly use non-A3R components in modular fashion (e.g., QT's SQL-based data persistence features) to replace their A3R equivalents (either initially or after an A3R-standalone protyping phase).

NCN and A3R in Consort

While NCN applications need not use A3R, or vice-versa, the two models are organically paired together. Their integration can take the form of NCN servers hosting A3R applications as resources,

and/or **A3R** software connecting to **NCN** instances as a domain-specific cloud back-end. The **A3R** metadata paradigm, based on "Hypergraph Ontologies", provides tools to streamline the encoding and distribution of application-specific data structures. This model thereby accelerates the process of implementing cloud services procedurally aligned with **A3R** components, because complementary **A3R** and **NCN** endpoints can share the same information representations. Moreover, **A3R** interface definitions can serve as references for implementing compatible **NCN** server-side code; the interface specification illustrates which client-side procedures will handle any server-originating data structures, so the server-side data providers can be constructed accordingly.

To ensure rigorous alignment between client and server endpoints, **NA3** employs a data modeling paradigm based on hypergraphs; the mathematical framework for the relevant new hypergraph model (which adds some additional structure to the theory of Hypergraph Categories) is provisionally outlined in a chapter of a book edited by LTS's founder (the chapter authored by a member of the LTS team), soon be published (we can share this material, or alternatively a more thorough unpublished explication, as desired). In practical terms, the advantage of this model is that **Qt**-specific data structures can be conveniently serialized and shared with or through cloud services; meanwhile the relevant data structures — and their interface and procedural requirements — are rigorously characterized, to support application testing, code-verification, documentation, and systematic User Interface development. **NA3** concretely operationalizes theories which have been advanced in the scientific-computing and knowledge-engineering communities toward a more conceptually refined and "multi-scale" Semantic Web. **A3R** extends "Semantic Web alternatives" such as Conceptual Space Markup Language (**CSML**) and Categorial Informatics, while also providing a self-contained **C++** Hypergraph library comparable in some respects to AtomSpace (part of the OpenCOG platform) or to HypergraphDB (Hypergraphs and Conceptual Spaces have been proposed in combination as a comprehensive foundation for computational semantics, notably in the article *Interacting Conceptual Spaces* which arose from an Oxford University reading group on Category Theory and formal grammar).

A good example of an **A3R** use-case is that of hosting scientific data sets. According to the emerging "Research Object" paradigm, scientific data should be published alongside code which ensures that subsequent readers and researchers have the tools they need to access, analyze, and visualize the data set, including double-checking statistical analyses and/or replicating experiments. With **A3R**, data sets can be self-contained Research Object "bundles" while still being provisioned with full-featured desktop applications tailored to their specific information profile. LTS is actively developing a framework (called DataSet Creator, or dsC) for building data sets that are paired with academic publications and with native software implemented in custom fashion for each data set (we can provide links to several data sets published as demonstration examples of this technology). The hosting and implementation of data sets along these lines offers concrete examples of **NA3** solutions and an opportunity to promote **NA3**-style development in the scientific and publishing communities.

As this use-case illustrates, our novel **NCN** model is based on rigorous serialization and interface-definition paradigms; in comparison, **Qt** Cloud Services tended to reuse structures more appropriate for non-native contexts (e.g. **JSON**), which arguably limited client-to-server interoperability. LTS's **NA3** model, by contrast, is combined with data modeling and serialization features that bring their own benefits to application projects over all; as such, this model does not only provide cloud-integration capabilities, but can be used as an overarching application-development framework.

Hypergraph Data Modeling and Requirements Engineering

A3R uses a so-called "Hypergraph Data Modeling Protocol" (**HGDM**) to describe and serialize application data. Hypergraphs are a flexible and expressive representation mechanism; this permits data to be marshaled with less boilerplate code than would be needed for more restrictive formats such as **SQL** or **RDF**.

HGDM's expressiveness is further enhanced by the introduction of a structuring element called "channels", which are aggregates of edges (by analogy to hypernodes being aggregates of nodes). Channels in this sense are a mechanism not previously developed in Hypergraph databases or libraries (channels aggregate edges — most often edges that are each incident to one hypernode — without likewise grouping their other vertices; in this sense channels are distinct from hypernodes, which collapse edges into a single hyperedge). The theoretical basis for channels is discussed in Chapter Three of *Advances in Ubiquitous Computing, Cyber-Physical Systems, Smart Cities and Ecological Monitoring* (previewed here). Channels are particularly useful when modeling procedure types, leading to a version of type theory formalized in a Hypergraph context (see *Advances in Ubiquitous Computing*, page 94). In **A3R**, channel notations can be employed to identify procedure types; similar notation is then used for an Interface Definition Language, describing interrelated groups of procedures with their corresponding types.

Aside from streamlining data serialization and deserialization, an arguably more substantial benefit of expressive data models (such as via Hypergraphs) concerns how precise data models document coding assumptions and requirements, which promotes the long-term maintenance of an application. **A3R** supports several Requirements Engineering features, including:

- Range-based numeric values, which prohibits data being constructed when it fails to lie within empirically meaningful ranges
- The option to use Universal Numbers for non-integer values, in lieu of floating-point types. (Universal Numbers are a mathematical representation developed by John Gustafson, which in many contexts are more precise than floats).
- An overall preference for employing application-specific data types instead of generic types like integers and strings. Using types which expressly model a particular empirical phenomenon facilitates Requirements Engineering insofar as the custom type becomes a basic unit of asserting and verifying requirements.
- A novel channel/hypergraph-based **C++** reflection mechanism, which can be used in addition to or in place of **Qt** meta-objects. This in turn promotes scripting and testing for custom data types, so that types' testing and implementation can proceed in modular fashion, with requirements observed at the individual type level.
- Leveraging **Qt**'s model/view architecture to pair up application-specific data types with corresponding **GUI** component types. In general, documenting inter-type relations — particularly the connections between data structures and their corresponding **GUI** representations — leads to rigorous application development and effective maintenance (see *Advances in Ubiquitous Computing*, page 55).

The Hypergraph Text Encoding Protocol

Correlated with **HGDM**, which uses hypergraphs to model raw data, **A3R** defines a Hypergraph Text Encoding Protocol (**HTXN**) for natural-language text. **HTXN** is relevant for **A3R** applications when users wish to compose manuscripts (e.g., technical papers describing research findings) related to data managed by an **A3R** application (although **HTXN** can equally well be used as a general-purpose document-preparation tool, outside the **A3R** context). The **HTXN** parsers and document generators are designed to be embedded in a host application; if this host recognizes the **HTXN** protocol, it is possible for **HTXN** documents to include instructions which call procedures exposed by the host application (so as to insert application data into a manuscript, for example). In general, **HTXN** is engineered to prioritize integrating and cross-referencing publications and data sets.

The benefits of **HTXN** include:

Mix-and-Match Formats Given documents encoded in **HTXN**, "secondary documents", or "views", can be generated in a variety of conventional formats, such as LaTeX and **XML**. **HTXN** employs "stand-off" annotation, which allows multiple markup protocols to be defined simultaneously on the same document. In addition, **HTXN** uses a flexible character-encoding protocol that ensures compatibility with diverse text representations (such as **XML**, LaTeX, Unicode, and **Qt**/QString). Overall, then, **HTXN** can be useful when it is desirable to employ different formats for a single manuscript at different stages of the publication workflow: for instance, **XML** for editing and LaTeX for **PDF** generation.

Fine-Grained Character Encoding The **HTXN** protocol includes more detailed document structure information compared to conventional markup. For example, **HTXN** identifies sentence boundaries and punctuation features, disambiguating logically distinct but often visually identical characters (such as dots/periods or dashes/hyphens, which have different structural meanings in different contexts).

A LaTeX-Compatible Document Object Model **HTXN** files have a graph-based structure that can be searched and traversed similar to the **XML** Document Object Model. **HTXN**'s document model, however, is designed to represent most structural features of LaTeX representation as well as **XML**. Therefore, in conjunction with generating LaTeX views on an **HTXN** manuscript, **HTXN** provides in effect a means to traverse LaTeX documents via software (e.g., via **C++** procedures).

A Transparent and Extensible Parsing Model **HTXN** is a text-encoding system, so it is impractical to compose documents in **HTXN** directly. Instead, **HTXN** files need to be built from text sources in some other format (**NA3** has its own markup style, called "**NGML**" for "Next-Generation Markup Language", which outputs **HTXN** and then LaTeX or **XML**). The **NGML** parsers can be readily adapted and modified, so that projects can use a custom-built document language if desired. The parsers and generators are also "transparent" in the sense that they are self-contained **Qt**/**C++** libraries that can be dropped into any **C++** project which can link against the **Qt** core (which also makes the components easy to examine through debugging sessions, when implementing alternative grammars or generators for customized markup languages targeting **HTXN**).

HTXN Plugins As a standalone library, **HTXN** (along with a document-composition language such as **NGML**) can be included in almost any application that is equipped to call procedures in **Qt**/**C++** libraries (obviously this includes scientific software written with **Qt** to begin with). Many scientific

and technical applications have a built-in plugin or extension mechanism, which makes it possible to introduce **HTXN** as an added feature; others are open-source projects where plugins can simply be inserted as new components in the source code. **HTXN** Plugins can then make **HTXN** available to authors who regularly use the host application for their scientific or research work.

Publisher-Specific HTXN When generating **XML** from **HTXN** documents, applications can target a specific **DTD** depending on the publisher to which a manuscript will be submitted. The resulting **XML** files can then be included alongside LaTeX and **PDF** in the documents sent to the publisher. This benefits publishers in turn, because it eliminates the need for a separate conversion process; submitted manuscripts will already have a version which is compatible with the remainder of their publication workflow.

The option of validating **HTXN**-generated **XML** against publisher-specific **DTD**s — combined with **HTXN** plugins to new or pre-existing scientific or technical applications — opens up the possibility of *publisher-specific* plugins, which can include **HTXN** but also other features of the publisher's platform (such as access to **API**s). In effect, publisher-specific plugins serve as miniature **NA3** applications embedded in some other software. Aside from practical document-preparation benefits, these plugins may also serve as a promotional vehicle for publishers — the plugin documentation, as well as a splash-screen when the plugin is loaded, could describe advanced features of the publishers' online and/or document-curation capabilities. Publishers are branching out away from conventional text documents to incorporate data sets and multi-media content; it behooves them therefore to find opportunities to describe and promote the more advanced features of their platform. Moreover, plugins demonstrate a level of technical sophistication; implementing plugins to scientific and technical applications demands advanced software-engineering techniques, so a publisher-specific plugin signals to the academic community that the publisher is comfortable engineering or designing complex scientific-computing components.

Using HTXN and HGDM In Consort

As hypergraph text and data encoding protocols, respectively, **HTXN** and **HGDM** can interoperate organically. The typical scenario where both formats would be used is that of preparing academic manuscripts accompanied by data sets which are serialized or documented via **HGDM**. Specific data samples, or figure illustrations derived from the data set, may then need to be inserted into **HTXN** publications. **A3R** components would be implemented accordingly to track text-to-data cross-references (and vice-versa).

A typical **A3R** data set provides native front-end code so that the shared data can be viewed and manipulated through custom **GUI** components. In conjunction with **PDF** manuscripts explicating or analyzing the corresponding data, readers then have two potentially interrelated components (typically two top-level windows), one for the **PDF** and one for the raw data. Ideally, the two windows would interoperate, e.g. via (in **A3R** parlance) "Coordinated Context Menus". In the canonical case, a context menu action activated on a particular sample, field, or other data set element (such as a table column) would load the **PDF** to the text location where that aspect of the data set is discussed. In the other direction, links embedded in the **PDF** can trigger a signal to manipulate the data set **GUI** in

correlation with the relevant text location — again the canonical case is scrolling and "unhiding" to ensure that a data sample or field, mentioned at that text point, is visible.

Case Study

To concretely illustrate HTXN/HGDM interop, consider the following scenario (essentially typified by one of the data sets accompanying *Advances in Ubiquitous Computing*). Consider a data set including linguistic corpora annotated with a grammar format such as CoNLL-U (named after the Conference on Computational Natural Language Learning). Assuming that some corpus samples are discussed in an article accompanying the data set, these samples are then found in two places: as numbered examples (according to linguistic convention) within the text, as well as in the raw data. Here is a good example of text/data cross-references: assuming that the GUI components for the data set are some kind of list, table, or tree view where users can scroll through the language samples, optimal integration implies that context menus associated with individual samples on the data end are correlated with locations where the corresponding sample is printed in the article text. This is an especially substantial case of cross-referencing because the actual raw data (sentences or other language/dialog excerpts) is fully present in the text (in the form of numbered linguistic examples). The essential step in this cross-referencing is to embed data in the PDF file which uniquely identifies each sample, to be read by customized PDF viewers so as to orchestrate the desired GUI coordination.

A further requirement comes into play if the text will include figures documenting samples' syntactic structure (perhaps generated by a LaTeX package such as TikZ-dependency). Building such illustrations requires extracting certain fields from the CoNLL-U data and using the selected information to populate LaTeX code inserted at the desired document position. This can be achieved by bundling a CoNLL-U library (such as "UDPipe") with the data set and implementing a C++ procedure to call the relevant UDPipe functions, build an intermediate data structure, and use the result to fill in a LaTeX template. This C++ procedure can then be triggered by instructions at the relevant locations in the article's HTXN file.

In this scenario generating publisher-specific XML may be important to help ensure that the data/text cross-references get preserved. If instead the publisher derives their XML files by conversion (e.g., from LaTeX), the LaTeX labels engineered to enable "Context Menu Coordination" may be eliminated or renamed, which can force a time-consuming task of reconstructing text anchors.