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. In **NCN**, the term "Native Cloud/Native" is used to describe hybrid applications whose server and client endpoints are both native components – 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. Meanwhile, OpenShift discontinued their free-tier Cloud/Native hosting last year, and another Cloud/Native service, Arukas, shut at the end of January 2020. 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, as one method).

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 (Linguistic Technology Systems) 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 **NCN** 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, or a minimal command-line "make" instruction. 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 (**UX**) 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 often opaque to the debugger — that is, it may be impossible in some contexts to examine their function bodies or stack frames.

Another benefit of using internal **A3R** components is that they can be simultaneously compiled into **NCN** instances developed alongside them. With that said, developers implementing **NCN**-based solutions 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 prototyping phase).

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 Hypergraphs or "Hypergraph Ontologies" (discussed in the next section), 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. More precisely, an **A3R** interface presents a "procedural contract" — it guarantees that a given suite of procedures are available to be launched, insofar as the application receives new data from a server. The server-side code should then be implemented to send data according to those contract-specified procedures' requirements.

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; a more detailed unpublished review of channel "theory" is also available on request). 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).

Hypergraph data models provide some of the same documentary benefits as Semantic Web Ontologies, only potentially more so, insofar as ordinary "Semantic" Ontologies have fewer resources to describe multi-tier structures (at least Ontologies in the conventional sense targeting single-tier graph structures, canonically those of **RDF**). The data and type representation architecture for **HGDM** aims to develop an expressive, general-purpose type system which is naturally suited to data serialization and which is compatible with main-

stream programming languages (we can provide more details about the space of type constructions recognized according to this system on request). In effect, almost any kind of function signature can be mapped to a corresponding HGDM-modeled type (technically, to a *channel complex*), and then used in an Interface Definition. Analogously, data structures which are instances of almost any type can be converted to a hypergraph structure, from where they may be automatically serialized.

Native Application Development in the Larger IT Context

While users have gravitated to web and phone applications for some day-to-day tasks (social networking, banking, e-commerce), desktop-style native software remains the preeminent front-end paradigm in sectors where complex, highly interactive GUIs are needed: science, engineering, industrial design, architecture, biomedical, pharmaceutical, military, and so forth. QT is the most popular framework among companies who develop native *cross-platform* software, as opposed to applications targeting a single Operating System. Supporting those developers, in turn, is a network of companies (many of them QT "partners") implementing GUI components, developer tools, QT Creator plugins, and other enhancements to the QT ecosystem. On purely technical grounds, it is reasonable to say — via its Cloud/Native, scripting, and data-modeling strategies — that NA3 can likewise be a significant addition to the QT ecosystem, assuming it is developed and stress-tested to production-grade maturity.

This document has focused on NA3's default (QT) implementation, although it is worth adding that the unique technical innovations expressed via NA3 have applications beyond the QT ecosystem. In most cases, QT data types and protocols have corresponding equivalents in other application frameworks, both cross-platform and Operating-System-specific, such as wxWidgets (cross-platform), Xcode (Apple) or MFC (Microsoft Foundation Classes). A reasonable estimate is that porting NA3 to non-QT platforms would comprise a six-month project for a two- or three-person development team. In order to stay focused on the near-term strategy for NA3, however, the remainder of this presentation will restrict attention to the QT-specific version.

QT has approximately one million active developers, over 5,000 client companies, and tens of mullions of downloads of recent QT versions (metrics according to the QT Group). These official figures may actually undercount the full QT market size; in particular, they do not appear to reflect open-source QT licenses (e.g., Linux software for the KDE/Plasma environment). QT is used for many scientific computing applications, in numerous disciplines; a representative sample includes CERN ROOT (CERN's physics/subatomic analytics platform), IQmol (for chemistry/molecular physics), medInria (for radiology), TeXstudio (for LaTeX processing), Mendeley Desktop (for Reference/Citation Management), QGIS (for Geoinformatics), MeshLab (for 3D modeling), OpenSCAD (for 3D geometry), Octave (a MATLAB emulator), ParaView (for data visualization), and the QT Creator IDE (Integrated Development Environment). Most of these applications would not be included in the QT company's official user metrics because they are maintained by academic or research institutions with an open-source QT license; nevertheless, institutions allocate resources for developing and maintaining technical applications. In sum, academic and Linux-oriented projects should be included in the commercial QT market, because organizations may contract for the implementation of QT software, even if the finished product will be released non-commercially.

A further consideration is that academic projects often split off to become commercial products; QT software used in academia may at some point be commercialized. This is not only a matter of software products themselves being marketed; it applies to any product (such as biomedical, military, Cyber-Physical, automotive, or industrial equipment) where QT software is essential to the operational or maintenance of the deployed product. In these scenarios, software originally built in an open-source context for Research and Development

then becomes an operational asset in a commercialized infrastructure. This is another dimension to the QT market space which may not be properly assessed when considering commercial QT licensees exclusively.

Part of the QT market, of course, is the QT ecosystem itself; commercially-licensed QT code bases or development tools get used in turn by QT-oriented companies to implement end-user solutions. QT "products" as such are either software applications for end users or developer tools via which the former are implemented (although components conceived for single projects may later be reused as general tools). History suggests that most commercial software products generate revenue, in their early stages, primarily from customer-specific customizations, but then eventually derive their most valuable profit-stream from commercial licenses. Special-license customizations help mold the product into a widely-usable standard version, given the natural feedback loop which emerges as the product's development team implements project-specific deployments, observing "in the field" which features are most useful and how these features are best made available to developers.

Taking the official QT partners cohort as a representative cross-section of the QT market, we can find companies whose revenue is driven by custom software development (ICS, Woboq, KDAB, Base2, Bitfactor, Sequality), by commercial licensing (Wind River, VNC Automotive, FrogLogic, NXP), by realtime and/or platform services (Mender, Timesys, Mapbox), as well as hardware/microprocessor providers (Toradex, ARM, Texas Instruments). It is probably true that companies whose products depend in whole or in significant part on QT generate revenue from customization and consulting more than in other technology sectors. This may reflect the position of front-end technology in relation to software in general: many software projects begin with new kinds of data or new user-interaction models, and only later address the need for implementing high-quality GUIs. Given that desktop-style front-end development is a rather specialized subdiscipline, many organizations end up hiring QT-focused companies as service contractors, which in turn supports a robust QT ecosystem (data from sources such as Glassdoor suggest that larger QT consulting partners have revenues roughly comparable to QT itself, indicating that the worldwide QT consulting/contracting market falls in the US $150-$250 Million range; factoring in commercial licenses, QT-enabled hardware, and QT-based software reasonably projects the overall QT market to roughly one-half billion US dollars). As a further detail, the "Workflow Management System" Market (where NA3 may be applied by virtue of inter-application networking) is projected to reach US $9.87 Billion (source: *MarketsandMarkets*).

This being said, financial records released by the QT company itself suggest that commercial ("Developer" and "Distribution") licenses are QT's largest revenues source (targets released in 2018 indicate that The QT Group Plc aims for 60% revenue from licenses, 20% from consulting, and 20% for "support and maintenance", which is an offshoot of developer licenses; total net revenue across these sources from 2018, the most recent figures available, was €45.6 Million, just over US $57 Million at 2018 rates).

It is premature to estimate a comparable partonomy of revenue share for NA3, but we can identify four distinct profit streams appropriate for NA3 as an integrated platform:

1. **Customization** Custom-implemented applications using project-specific versions of NCN and/or A3R.
2. **Licensing** Commercial licenses required for any deployment of NCN outside LTS-controled servers and/or any deployment of A3R applications (or of software including A3R components for such development requirements as databases, data modeling, scripting, data serializing/deserialization, and text parsing) in a commercial context.
3. **Hosting** LTS anticipates running proprietary containers via a Cloud-Native service such as OpenShift, and then leasing access to this service to NA3 users. LTS can offer integrated hosting and consulting wherein LTS fully implements and maintains a back-end paired to any desktop/native client software. Because the

expertise involved in building native desktop applications is very different from the techniques required to deploy a Cloud-Native container image, the option of delegating all backend responsibilities to LTS may appeal to QT-oriented development teams.

4. **Sponsorship** As discussed below, LTS anticipates running a data-sharing platform which would be a publicly-visible introduction to LTS's in-house NCN service (whereas other sub- or para-containers would be leased to third parties and provide publicly-visible content only at their discretion). This "demo" container, while being a vehicle for the general public to learn about NA3, would also host research data sets and would therefore be a resource in the public interest, allowing LTS to receive compensation from companies financially supporting the portal because of its merits as a technology benefiting science or scholarship.

The remainder of this summary, to elaborate further on the hosting and sponsorship possibilities, will focus on cloud services expressly maintained by LTS (in contrast to commercial NCN instances whose licensees host the NCN code on their own). In NCN parlance, sub- or para-containers are units within a larger NCN environment utilizing isolated HTTP access protocols and data/file storage. Each such partial container can be twinned with a specific A3R application, providing a cloud end-point for storing application-specific data and/or sharing such data between different executable instances of the application. Potentially, then, any A3R project developed by LTS may have a corresponding presence on LTS's cloud resources.

At the same time, the A3R model also envisions desktop applications as self-contained, shareable units, which can be hosted on web servers (including NCN instances) as zip and metadata files. Therefore, LTS's NCN deployment can serve as an access-point for users acquiring or obtaining information about A3R applications (including data sets published as "Research Objects" using A3R within their code base). The next three sections will discuss these academic applications further.

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 Once a document has been constructed in HTXN — this becomes the "primary" document — a choice of generators is then available (or can be implemented) to produce secondary documents, or "views", in a variety of conventional formats (canonically LaTeX and XML). HTXN employs "stand-off" annotation, which allows multiple markup protocols to be defined simultaneously on the same document. Accordingly, an HTXN document may be prepared for eventual use by both LaTeX and XML generators, including XML content (such as attributes) ignored by the LaTeX generator, and vice-versa. 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 necessary 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 similarly to the XML Document Object Model. HTXN's document model, however, is designed to represent most structural features of LaTeX encoding 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, for projects to 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 may 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 could 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 DTDs — combined with HTXN plugins to new or pre-existing scientific or technical applications — opens up the possibility of *publisher-specific* plugins, which may include an HTXN implementation, but also other features relevant to the publisher's platform (such as access to APIs). In effect, publisher-specific plugins serve as miniature NA3 applications embedded in some other software. Aside from practical document-preparation benefits, these plugins might 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: A case study

As hypergraph text and data encoding protocols, respectively, **HTXN** and **HGDM** can interoperate organically. The canonical 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 or unhiding elements, as necessary, to ensure that a data sample or field, mentioned at that text point, is visible. Context Menus can also be coordinated among three or more windows (for example, between viewers for **PDF**, **3D**, and **2D** charts or diagrams).

To concretely illustrate **HTXN/HGDM** interop, consider the following scenario (this is a situation typified by one of the data sets accompanying *Advances in Ubiquitous Computing*, which can serve as a case-study). 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: first as numbered examples (according to linguistic convention) within the text, and secondly 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 with the data set, such as **UDPipe** (which is written in **C++**). Then, a **C++** procedure could be implemented to call the relevant **UDPipe** functions, build an intermediate data structure, and use the result to fill in a LaTeX template. This **C++** procedure may 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 labels engineered to enable "Context Menu Coordination" may be eliminated or renamed, forcing a time-consuming task of reconstructing text anchors.

NA3 and the Research Object Protocol

Open-access data sets conforming to the Research Object protocol are a good example of use-cases where the A3R development strategies may be beneficial. According to the Research Object protocol, data sets are paired with code and metadata to help subsequent researchers use and interact with the published data. Via A3R, the Research Object can be implemented as a standalone, desktop-style "dataset application" whose data models and GUI components are uniquely designed for the associated data set, reflecting its scientific and theoretical provenance (experimental setup, data-acquisition methodology, data-structural rationale, etc.). A3R employs unusually rigorous modeling for application components and data types, which makes it particularly appropriate for this Research Object context wherein a data set's technology — its structural organization and custom code base — becomes itself a scientific artifact.

Academic data-hosting is also a sector where LTS has a marketing head-start, insofar as LTS founder, Amy Neustein, Ph.D., serves as Editor-in-Chief of the *International Journal of Speech Technology* and has authored/edited 12 academic volumes. In addition, she serves as editor of three book series, adding up to more than 50 academic and technical books. LTS is currently in discussions with several publishers to make A3R tools available to authors for document and/or data-set preparation. LTS is pursuing such collaborations partly to introduce NA3 within the scientific community and partly to curate and spur the emergence of an A3R application corpus.

A3R data sets serve two distinct purposes in the context of marketing NA3. On the one hand, these data sets may be published on respected scientific platforms (notwithstanding their also being hosted, potentially, on an NCN service), which provides a forum for promoting NA3 to the scientific, academic, and IT communities (through included A3R code as well as documentation that will explain both the NCN and A3R frameworks). Second, open-access A3R data sets model a specific non-commercial version of A3R, which serves as a baseline demonstration of A3R features. The open-source A3R implementation provides rudimentary support for scripting, data persistence, cloud integration, 3D graphics, embedded web viewers, GUI-based unit and integration testing, multi-application networking/workflows, and other features often desired for contemporary application development. Licensee developers (or LTS itself in a consulting/contracting role) can then extend whichever of these features are relevant for a commercial project. The minimal dataset applications serve to prototype the overall structure of A3R software, helping developers and/or acquisition teams visualize the practical benefits of A3R and also decide on which A3R features they will use on a commercial-grade scale in their project.

Generalizing to NA3 overall, the concrete example of A3R data sets helps illustrate NCN features, because server-side capabilities are best understood in terms of how they complement client-side UX. Desirable cloud-integration features include persisting one user's application state across computers (home/school/office, etc.); sharing data in application-specific formats between users; collaborative editing; non-local backup; web-searching application corpora; and dynamic or non-recompile upgrades to running applications. Although such capabilities in the context of NA3 depend on properly implemented NCN services, they can be tangibly demonstrated for prospective customers through prototypical A3R front-ends.

In this sense open-access data sets created via A3R serve as demonstrations and concrete examples of NA3 technology, and an opportunity to (indirectly) market NA3 in the relevant scientific, publishing, and computer-scientific communities. Currently three A3R data sets have been published in the fields of linguistics, speech technology, and biomedical Cyber-Physical systems. Several additional data sets will be made publicly available in conjunction with the upcoming publication of *Advances in Ubiquitous Computing*.

LTS can therefore arrange to demonstrate the various facets of NA3 using published data sets as a case-study. We can supply additional references, or more detailed information about NA3 components and technical innovations, as desired.