Enterprise Integration Modeling

A Practical Enterprise Integration Solution Featuring an Incremental Approach via Prototyping

Mihaela Iridon
Cândea LLC
Dallas, TX, USA
e-mail: iridon.mihaela@gmail.com

Abstract— **As larger and more complex line-of-business (LoB) software systems emerge and grow within an organization so does the need for such systems to interact with each other and exchange data, making it imperative to design flexible, scalable integration architectures and frameworks to support a robust and well-performing enterprise system. System integration is a multi-faceted undertaking, ranging from low-level data sharing (Shared Repository or File Sharing), to point-to-point communications (Remote Procedure Invocation via Service Orientation), to decoupled data exchange architectures (Messaging). It is not uncommon to build entire integration sub-systems responsible not only for exchanging information between systems (commands and notifications) but also for potentially more complex business logic orchestration across the entire enterprise (Message Broker). Moreover, implementing large integration solutions carries a considerable amount of risk so it is imperative that such solutions be validated by releasing functional prototypes to a smaller client bases in order to ascertain the benefits of - and perhaps the clients' interest in - delivering new features. This paper is contemplating a practical data notification and synchronization integration solution that allows multiple enterprise domains to share data that is critical for business operations. The solution features an incremental delivery approach based on initial prototyping that allowed for additional market analysis and a gradual integration. The article presents the architecture achieving this business objective, together with the corresponding system models and design artifacts. It described the data integration solution realized using a broker-based messaging approach employing various enterprise integration patterns, as well as the initial synchronous functional prototype and the many benefits of software system prototyping in general.**

Keywords-enterprise integration; system modeling; data integration; canonical model; integration patterns; prototyping; simulation; testing.

I. INTRODUCTION

Within an enterprise, system integration solutions are usually designed and implemented as an afterthought, as an attempt to build or to expand a new or existing enterprise architecture comprised of heterogeneous legacy system. It may be safe to say that most companies do not start with an integrated enterprise architecture but rather a core domain (also referred to as a vertical), which will eventually grow and become part of a larger enterprise system as the industry case described in [1]. In many cases, such integration is achieved by employing various off-the-shelf integration products, such as Microsoft's BizTalk [2] or TIBCO's Integration Platform [3].

Software system integration comes in different flavors, depending on the business objectives, the overall enterprise architecture, and ultimately the realization approach chosen. In Section II, we will investigate these driving factors and then present a concrete implementation approach and its models in Section III, as it has been proposed and adopted by a provider of the nation's largest portfolio of benefit and payroll products and services designed to help more than 200,000 small and medium-sized businesses [1].

The beginning of Section III also examines the motivation behind this paper by attempting to set the right expectations with the reader and to rationalize the purpose of the technical artifacts gathered here. It strives to provide relevant context and comprehension that underscores the focal point of this document: a practical application of integration patterns and system integration modeling towards building a concrete industry solution, with the intention of sharing experience, approaches, challenges, and design artifacts that are neither trivial nor stereotypical.

The present article is an elaboration of the "Enterprise Integration Modeling: A Practical Enterprise Data Integration and Synchronization Solution" paper presented at IARIA's First International Conference on Fundamentals and Advances in Software Systems Integration (FASSI 2015) [1]. This paper focuses on architectural modeling applied in a real-case enterprise implementation, but also captures relevant aspects regarding prototyping as a tool for analysis and risk mitigation that enabled a phased market release.

The central topic described in this paper represents a data integration and synchronization blueprint aimed at implementing the "Maintain Data Copies" data integration pattern [4] by means of a decoupled integration mechanism realized on a custom broker-based messaging architecture [1] [5] [6]. The data payloads exchanged between the loosely coupled sub-systems abide to a *ubiquitous* integration language, referred to as the *canonical model* [7] and is described in Section IV. This model is the unified abstraction of the data structures that must be shared and synchronized between these systems.

Section V describes the functional prototype that was initially implemented and released to a reduced client base. It features a synchronous messaging approach as a generalization of the larger integration vision. The purpose of the prototype was to provide the necessary tools for a deeper

analysis, both of the market reception and feature usability, as well as of the overall integration challenges and effort.

Section V concludes by presenting various aspects and benefits of software system prototyping – as identified in this particular implementation – with emphasis on prototyping for system integration. It also discusses how prototyping and building synthetic components helped alleviate some of the challenges encountered, including distributed teams' collaboration, component development, and unit and system integration testing.

Concluding remarks and highlights of the information shared in this paper are summarized in the final section.

II. System Integration Perspectives: Comparing and Contrasting Functional and Data Integration

When building a large enterprise software system by bringing together multiple domain applications, it is important to first identify the level of abstraction at which the integration specifications are being defined: Do the integrating sub-systems only need the data that allows them to carry out their own functions, or do they also require access to cross-domain exposed functional features? In other words, should a system expose data only or features as well?

The answers to these questions will determine the type of integration that must be realized: data or functional integration, and, perhaps even further, it will help discern between the need of a flexible, lightweight, loosely-coupled integration architecture and one that adds enterprise features and interactions, transcending domain system boundaries.

It is also possible that, in some cases, a hybrid approach will be pertinent, either to realize a quick and simple integration with a narrower scope (e.g., a pilot or test product implementation), or to overcome deep architectural and data model discrepancies between the existing systems. In this case, the solution must fulfill some imperative enterprise needs - whether they are related to exposing new system features in a short amount of time or at a lower cost until further market research proves the worthiness of additional funding for a comprehensive, scalable, extensible, and suitable solution. These considerations are primarily relevant when contemplating a phased delivery to the customer base in order to reduce the amount of risk that larger, more complex integration solutions usually incur.

A. Functional Integration

This type of integration involves exposing data and behavior [8] to systems that participate in the integration in order to trigger or invoke business features exposed by these systems. Usually, a pure Service Oriented Architecture (SOA) [9] [10] would be the simplest architectural approach that could realize this requirement, but it would introduce system coupling and would also lead to serious scalability concerns [11]. However, a synchronous point-to-point integration solution is perfectly suitable in many cases, and – as it will be presented here – would make perfect candidate for an initial system prototype. Web Services implement in effect the Remote Procedure Invocation integration pattern paradigm [7] and this implies mutual awareness of the presence of – and the functionality provided by - each of the integrating systems.

Complexity becomes apparent when more than two systems must interact at a logical and/or functional level of abstraction by invoking these exposed features and generating chattiness across the network, or when systems evolve, possibly threatening the stability of the integration contracts and hence of the solution. Several options are available to alleviate these problems, from architectural ones to following best practices, proper functional decomposition, and service encapsulation, and eventually to making the proper technology choices [10].

B. Data Integration

This type of integration assumes that the various integrating systems were not designed to work together [12], and that they do not have direct access to the entire enterprise data but only to that which they provision directly. These systems were built in order to fulfill certain functional and business requirements, rather than architectural ones. It is also possible that some systems were acquired later (e.g., corporate mergers, third-party software acquisitions, etc.)

Given that the systems evolved independently, enabling them to interoperate using multiple copies of the enterprise data (i.e., multiple data sources) while providing enterprise-level business features in a unified fashion is problematic, since there is no single source of truth and, potentially, no single source of data entry. Multiple applications may allow users to enter the same type of data from different user interfaces that sit atop of different business/logic layers and, consequently, different data sources.

Achieving this type of data integration can rely on either the delivery of custom solutions (for example, involving an enterprise service bus), or commercial tools (such as implementations of a Master Data Management system), which may expedite the time-to-market of such an integration, in some cases at lower costs than custom solutions [7] [13].

III. A Practical Data Integration and Synchronzation Solution

A. Setting the Expectations

1) This Paper Is Not a Comparative Study Including Integration Solutions and Approaches

The solution described in this paper is an actual integration design created for a client that had very specific requirements for bringing together a couple of business verticals and lay the foundation for adding a new vertical to the mix. The integration involved both legacy systems as well as a newly released one, and presented unique challenges that required extensive analysis and prototyping before the final custom solution was considered as a viable candidate. Enterprise integration always caters to very specific needs, as unique as the systems that they attempt to bring together.

This paper does not compare the solution designed for this particular client with other enterprise integration solutions but rather focuses on a particular implementation for an actual client who elicited this solution and who delivered the initial prototype to their client base. Some of the reasons for not pursuing a comparative study against the solution presented here are described next.

Although at a high level architectural styles may be compared and contrasted, including the technologies employed, it is rather uncommon to come across detailed technical specifications of actual integration solutions from various industries to conduct such a comparative study. In some cases, enterprise system architecture is shown as very high-level, in the form of block diagrams, to the extent that they are relevant from a Business and/or Sales view. Component diagrams, architectural layers, interfaces and frameworks are usually handled as proprietary technical documentation and artifacts rather than being exposed for public evaluation.

It is also understandable why such artifacts are not publicized. Unless the company's software is going to be acquired by some other entity, the internal architecture of that software system is not necessarily relevant to potential consumers. Instead, its exposed features –functional and perhaps non-functional – are shared – up to a certain degree of detail. From a client's perspective, a software system – especially one that sits behind a service interface, will be treated as a black box whose features are exposed via rich user interfaces - web applications in most cases – whether the software system is self-hosted (by the client's infrastructure) or vendor-hosted, cloud-based or on premise. An adequate level of technical detail for other industry implementations is rather scarce and difficult to come by in order to assemble a meaningful comparison of behavior and/or performance.

One last important fact that prohibits the development of a proper and relevant comparison analysis of integration solutions is that various companies, even if they cater to similar domains (e.g., payroll service providers), they may adopt highly specialized system models and architectural approaches to building their enterprise integration, as dictated by the features they provide to their clients. Not uncommon, dealing with legacy systems is also a relevant aspect of modeling new features and/or adding integration capabilities to existing domains, since the complexity of such tasks increases significantly. This is primarily due to the difficulty of bringing together old and new technologies and practices. In any case, specific domain models are not usually shared openly; and they may vary greatly from one enterprise to another, despite the realization of similar functionality.

For this reason, any meaningful comparison with such enterprise systems would not carry a significant value, assuming that the specific solution's topology details and/or performance specs are disclosed and available for evaluation.

2) This Paper Does Not Introduce Groundbreaking Ideas for Solving Integration Problems

Many books and online resources on software system design and software system integration are very useful tools for understanding the many ways in which one can build robust, extensible, maintainable, scalable software [5] [7] [9] [11]. Patterns, principles, and best practices are always emphasized and more or less extensive examples are provided. However, usually such books have a very precise and well-organized agenda that they follow, introducing and/or elaborating on various technologies, architectural and/or integration styles, leaving it up to the reader to absorb all that knowledge and apply it in ways that are best suitable for the system that they are building. Rarely, if ever, is there a "one size fits all" approach to software design. Nevertheless, it takes skill, experience, and a good understanding of the problem and the domain to devise the appropriate architecture, layers, components, and how they interact with each other to build the system that is required. Moreover, in many if not most cases, architects and technical leads deal with various departmental, organizational, and technological constraints that may render the best solution unfeasible.

What this paper shows, however, is how various approaches, practices and industry recommendations were selected and chosen to build a practical solution for a client with clear requirements and constraints, that would enable their isolated business domains to share data.

B. The Businss Domains

Consider three major business domains, Human Resources (HR), Payroll, and Benefits. The common ground for all three is the demographic data that defines the companies (or clients) that these systems are servicing and their employees. As is quite often the case, neither domain was built with a true enterprise vision in mind, neither architecturally, nor functionally. Yet the main enterprise data on employees and clients served must be shared across all domains when multiple data copies exist, one per domain. These data sources were designed for a very specific purpose, making it prohibitively expensive to refactor the systems' layers and the business applications so that they rely on a single source of truth – a unified data source across the enterprise. A solution employing Master Data Management (MDM) tools has been evaluated but the business requirements did not warrant such elaborate implementations for this particular case. The proposed and agreed upon solution was to implement the "Maintain data copies" data integration pattern [4] by means of a custom scalable and extensible middleware architecture (or integrating layer [5]), reusable frameworks and models, and carefully-chosen technologies, to fulfill the business need of providing multiple services (HR, payroll, and benefits) to an array of small to large size clients.

The following sub-section presents the main models of the proposed integration solution, where data notifications are being exchanged between the various domains via a broker-based messaging architecture, using various enterprise integration patterns, also depicted later in the EAI pattern-mapping diagram in Figure 4. The data payload for these messages is wrapped inside a context-based notification model, allowing participating systems to take the appropriate action – based on their own domain rules – using the data received from the message broker. The individual domain systems are not aware of each other, only of the message broker through which they communicate.

C. The Structural and Behavioral Integration Models

All models, structural and behavioral, included in this paper are excerpts from the technical design specifications document created on behalf of the client's Enterprise Integration Solution [6] and they are being used hereby with permission from this client.

1) Structural Models: High-Level Enterprise Integration Architecture and Components

The integration middleware was designed as an extensible, highly responsive, and scalable broker-based topology through which the formerly isolated domain systems will exchange data notifications in near real-time and in a loosely coupled fashion. The middleware is built on durable messaging frameworks, such as an enterprise service bus (ESB), queues, an entity mapping/correlation infrastructure, and various service endpoints (SOA).

The high-level component diagram (Figure 1) shows the three business verticals as clients to the enterprise services that provide access to features that implement cross-cutting concerns (logging, SSO, audit) while indirectly exchanging data notification messages among each other.

This message exchange is intended to take place without reciprocal system awareness or knowledge of the features they provide, using the integration middleware exposed via a service endpoint (i.e., the Data Notification Receiver Service).

This design ensures system scalability and plasticity of the integration scope (data or functional), while hiding the actual technology specifics from the participating systems.

2) Object/Data Models: The Canonical Model

The data notifications exchanged between the systems via the service-broker integration middleware are structured as a two-layered object model. One is the actual data payload represented by the integration ubiquitous model, also referred to as the Canonical Model [7], and the second is the notification model which is wrapping (or encapsulating) the canonical model payload, adding context, source, and target details to the communication messages.

This allows for a reusable notification model, where - by employing generic data types for the payload wrapped within the notification together with the appropriate inheritance (generic type inheriting from the non-generic type) – we can design any number of notification schemata that could encapsulate any business entity models inside a generic payload. The payload is domain-specific (or enterprise integration-specific in this case), whereas the notification model is domain-agnostic. This is depicted in the object model in Figure 2. The generic type T of the payload can be represented by any domain entity. Section IV describes the standalone object model used for the enterprise integration solution presented here.

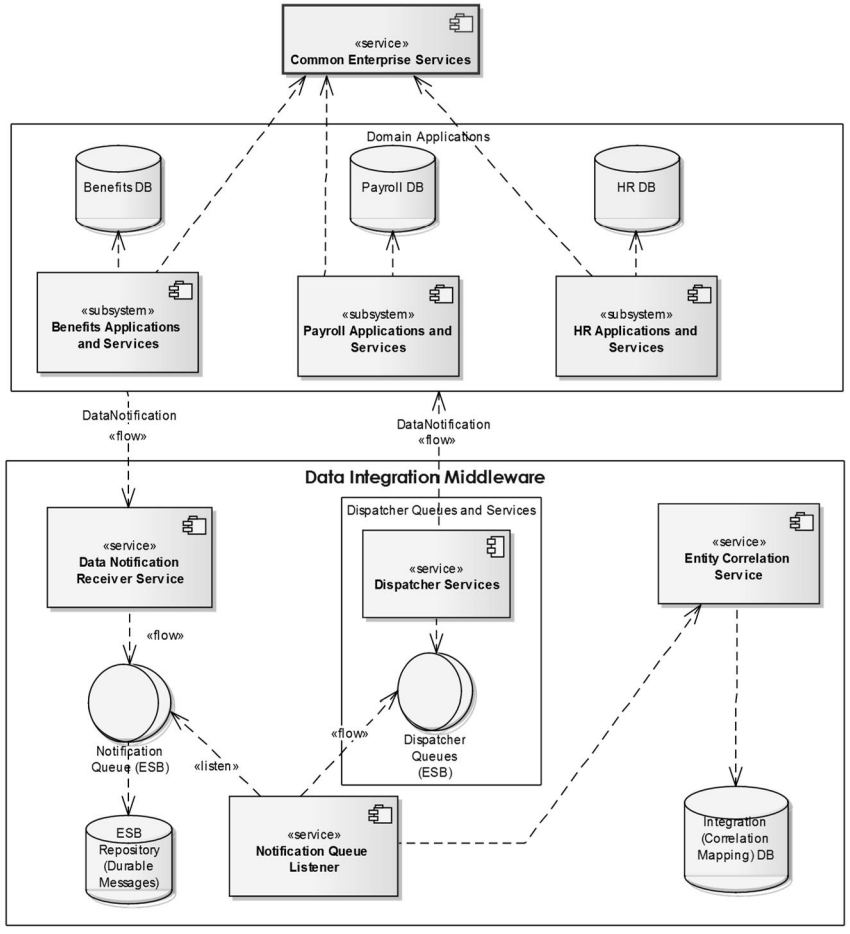

Figure 1. Overall enterprise integration topology: business verticals and integration middleware

This allows for a reusable notification model, where - by employing generic data types for the payload wrapped within the notification together with the appropriate inheritance (generic type inheriting from the non-generic type) – we can design any number of notification schemata that could encapsulate any business entity models inside a generic payload. The payload is domain-specific (or enterprise integration-specific in this case), whereas the notification model is domain-agnostic. This is depicted in the object model in Figure 2. The generic type T of the payload can be anything that one would define for a given domain: employee, client, address, benefit, participant, dependent, etc. In fact, a separate object model for the enterprise integration has been defined and is used in the implementation of this solution (see Section IV for further details).

3) Behavioral Models: The Communication Model Describing the Enterprise Data Synchronization Process

For the implemented solution, the data notification exchange follows a very simple path through the hub-and-spoke (or star) integration middleware topology (Figure 3). However, the main challenge that had to be overcome is associating the business entities from one system to business entities in other systems, without introducing direct dependencies between these systems or awareness of other domains or domain-specific identifiers that – semantically – tie these enterprise entities together. For this purpose, an entity correlation service was introduced, using a separate repository of entity IDs that represent logically - or semantically - identical entities across the enterprise. Such correlations will be specified during an initial data setup process by administrative users or via custom automation tools and import/export facilities.

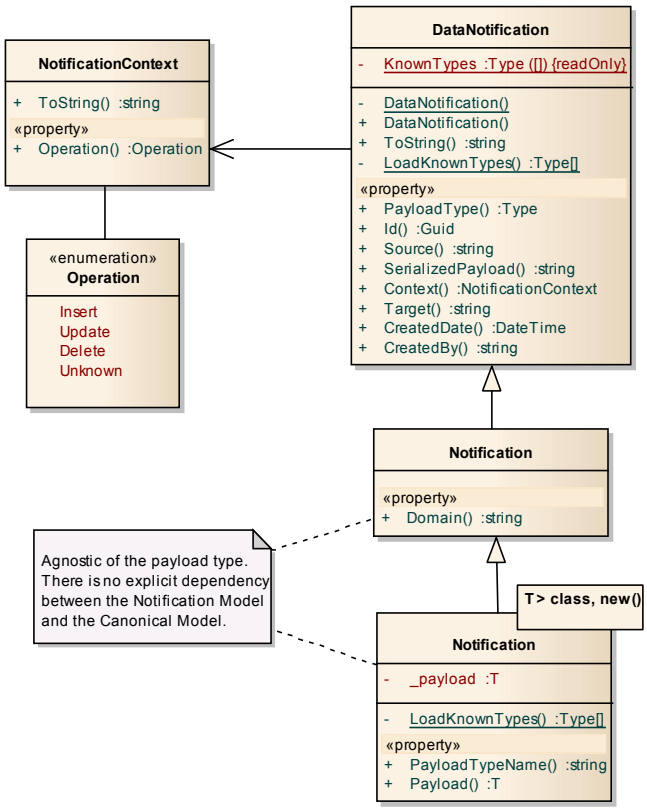

Figure 2. Data notification object model

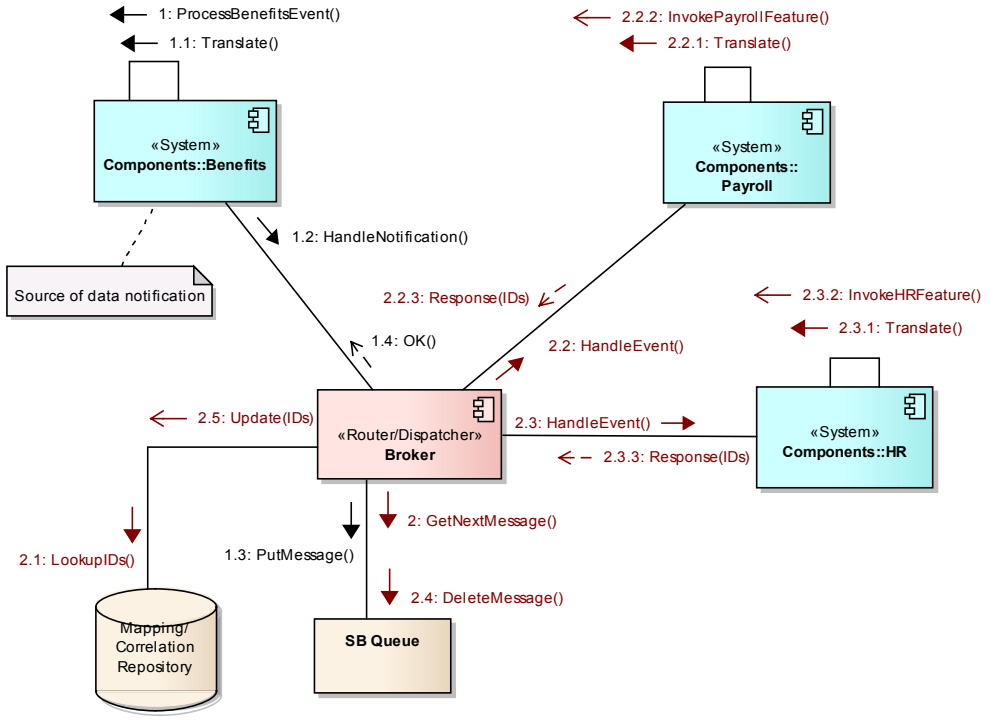

Figure 3. High-level integration communication model mapped to the service broker (star) topology

D. Integration Architecture Feature Highlights

Noteworthy features of this integration solution are compiled below. They are grouped into functional and non-functional characteristics. Several design details are included to impart to the reader additional context and comprehension of the architectural and technical approaches chosen.

1) Key Functional Attributes

a) Enterprise Data Coherence

Maintaining multiple data copies synchronized, all integrators become symmetrical systems of record for the core/common enterprise data.

All systems participating in the integration are able to notify the enterprise about relevant data updates in a particular line of business system without being aware of the other systems that might need this information or of the way in which this data will be consumed. They will do so by raising notifications with the integration middleware alone.

Consequently, the systems will be notified of relevant data updates occurring across the enterprise by receiving such notifications that encapsulate data payloads following a normalized model. These notifications are dispatched by the integration middleware, potentially based on specific integration rules and constraints.

This notification mechanism will in turn allows the integrating systems to keep their own data copy synchronized with the data across the enterprise, while continuing to provision it independently, according to the domain's business rules.

b) Enterprise Functional Coherence

Specialized domain services offered to clients will continue to be managed and augmented within each individual vertical, without the need to cross domain boundaries, since all necessary data is available at the domain level, nearly real-time consistent with the enterprise data.

Decoupled and asynchronous notifications exchanged via the messaging broker keep systems unaware and independent of each other, while allowing the enterprise to grow as needed. Additional applications may be added; if these applications require their own data copy, they will start listening to notifications from the middleware services. If they also support or require data updates that must be synced with other applications' data sources, then the new participants will also start sending notifications to the broker to be dispatched and consumed throughout the enterprise, as needed.

2) Key Quality Attributes

Large integration undertakings - as the one described here – can carry significant risks, require substantial effort to realize, and are built with a very long-term plan in mind. For this purpose, multiple non-functional features of the proposed solution have been identified and analyzed. A subset of all those considered with the client, mainly the critical ones, are presented next.

a) Scalability

Without any architectural changes to the integration framework or the domain systems, new systems can be added to this topology and can be enabled to participate in the integration (assuming they also use their own data source(s) that require continuous or occasional synchronization with the enterprise data).

The two main requirements for these systems are (a) to expose a data notification service endpoint that will handle enterprise notifications from the middleware (i.e., to react to notifications from the broker) and (b) to have the ability to raise such data notifications appropriately, while being aware of the canonical model as the lingua franca of the enterprise integration.

b) Testability

Although additional testing frameworks for the integration components must be designed and built, individual systems will continue to be tested independently of each other or the integration middleware.

Components that simulate/generate notification traffic through the integration framework can be built to allow for independent testing of the service broker and the integration infrastructure.

c) Maintainability

The basic SOLID design principles employed, and most importantly the "separation of concerns" (or SoC) principle, ensure a highly maintainable architecture and codebase due to overall high cohesion and low coupling [5] [11].

Domain rules do not escape the boundaries of the system to which they belong, and similarly integration logic is isolated to the broker components and services.

d) High Availability

By employing load balancing and clustering around the integration services and the choice of technology (e.g., Service Bus Farm), the deployment topology was designed to ensure high availability as far as the integration components are concerned.

e) Performance

Assuming appropriate technology choices, the integration framework ensures a high throughput of notifications with minimal integration logic (i.e., entity correlation map lookup) required between the moment of receiving a notification and that of dispatching one.

For example, Microsoft's Windows Server Service Bus 1.1 (on premise) can process 20k messages/second (based on 1K message size) with an average latency of 20-25ms [14].

f) Stability

The integration middleware and the canonical model had to be built in such a way that the overall system would not require changes over time. Moreover, the middleware had to be impervious to individual client failures. For this reason, a lot of thought and design hours were spent on the various models presented here, so that they can withstand various changes (and potential failures) of the integrating systems.

E. Enterprise Integration Patterns Mapping

Hohpe and Woolf compiled an excellent collection of asynchronous messaging integration patterns in their book [7]. Furthermore, their practical advice on designing such integration systems and the various examples provided helped with the design of this messaging architecture, while it also facilitated the selection of the appropriate topology and

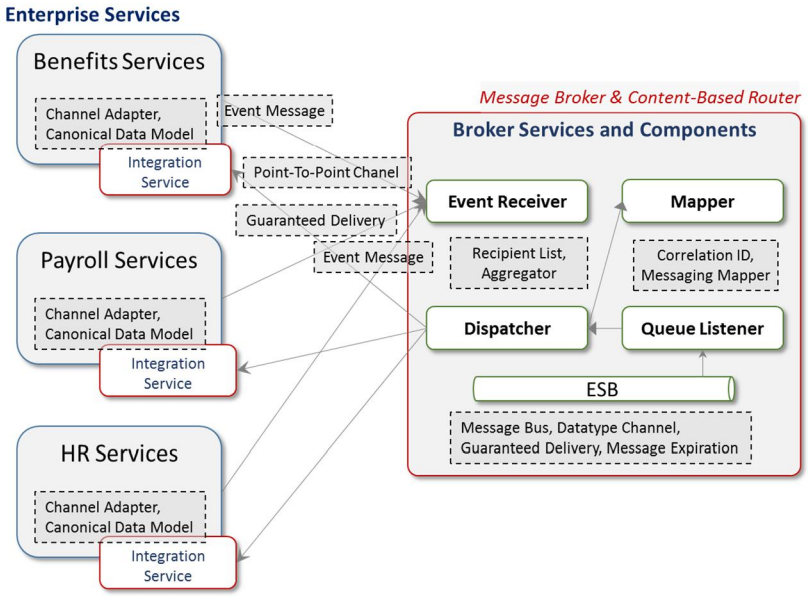

Figure 4. Mapping of enterprise integration patterns to domain systems and to integration middleware components

patterns that were fundamental to the delivery of an effective system integration solution.

It is interesting to see how the key integration patterns employed in the design and realization of the integration architecture directly map to the business verticals and integration middleware components. This mapping is shown as an overlay atop the simplified enterprise system block diagram in Figure 4.

IV. SUPPLEMENTAL INTEGRATION MODELS

A. The Canonical Model's Base Class Details

The Canonical Model integration pattern [7] has been the central theme of the solution implemented and is the only integration element that was allowed to permeate the enterprise (at each system's integration endpoints). This model can be envisioned as the ubiquitous integration language, which describes entities that are shared across the various domains of the enterprise. However, these entities in turn share data elements that are best modeled separately, as properties on base classes, using elemental inheritance, aggregation, and composition modeling concepts. For the domains in the presented case study, the need to support entity identifiers of different types, active timeframes, and traceability/audit features, led to the design of the model in Figure 5 where all domain entities inherit from the abstract class *EntityBase* shown in the center of the class diagram.

B. The Canonical Model and the Main Integration Entities

The main (aggregate root) entities in the integration's lingua franca are Group and Employee. They reflect the primary integration objective: keep Employee and Group demographics data in sync among all enterprise systems, by allowing each system to maintain and operate on their individual copy of the data. The model shown in Figure 6 is specific to the integration solution proposed for the client, aiming at integrating Benefits, Payroll, and Human Resources domains, more specifically for achieving the business goal of cross-selling services to various clients.

Noteworthy here is the fact that if we consider the canonical model as the domain of the integration, then it is following the anemic domain model design anti-pattern [15]. This is because these are simple data containers and do not encapsulate functionality as the integration framework's domain itself is behavior-less. The model's only purpose is to capture and transport data notifications across systems –so, from this (proper) perspective the model is abiding to the Data Transport Object (DTO) pattern of enterprise application architecture [11].

Generic functionality is exposed in the form of service operation contracts for handling notifications (whether a domain system raises a notification or must handle one), but no enterprise features are being implemented here, hence data representation and modeling is of essence and imperatively affects the success of the proposed system integration solution.

C. The Integration Activity Model

The overall system integration flow is modeled in the activity diagram in Figure 7, where the various integrating systems and the broker components are bounded by the vertical swim lanes, to indicate where activities and actions cross system boundaries. The diagram also shows how the correlation service is being employed to allow the integration framework to associate the same (logical) clients across domains by looking up and populating the appropriate domain identifiers, as part of the context that wraps the notification data payload passing through the broker.

Behind the broker services, multiple queues were utilized as a durable and priority-based messaging mechanism, in order to decouple the various processes that take place at the integration framework level: receiving messages, processing notifications, and dispatching them to targets.

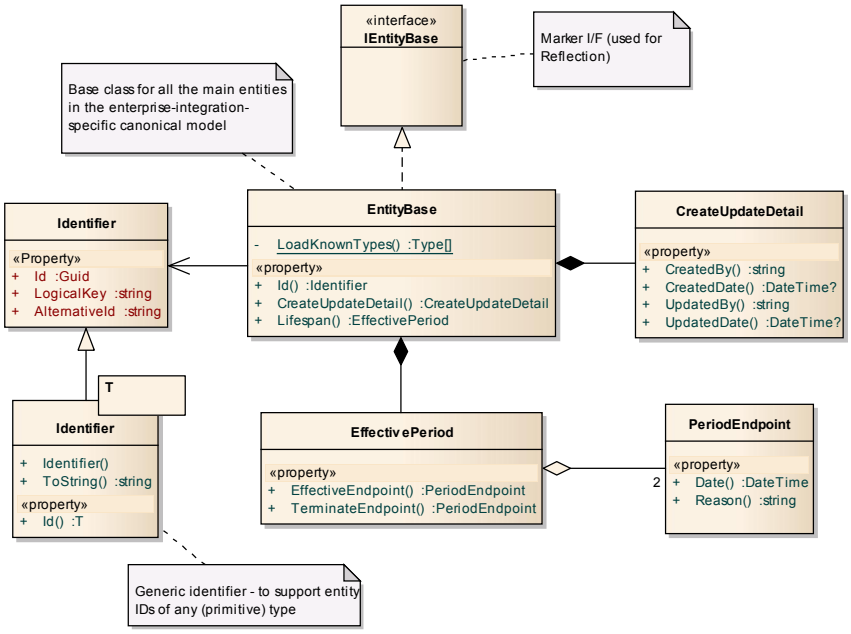

Figure 5. Base class and common elements for the canonical model types

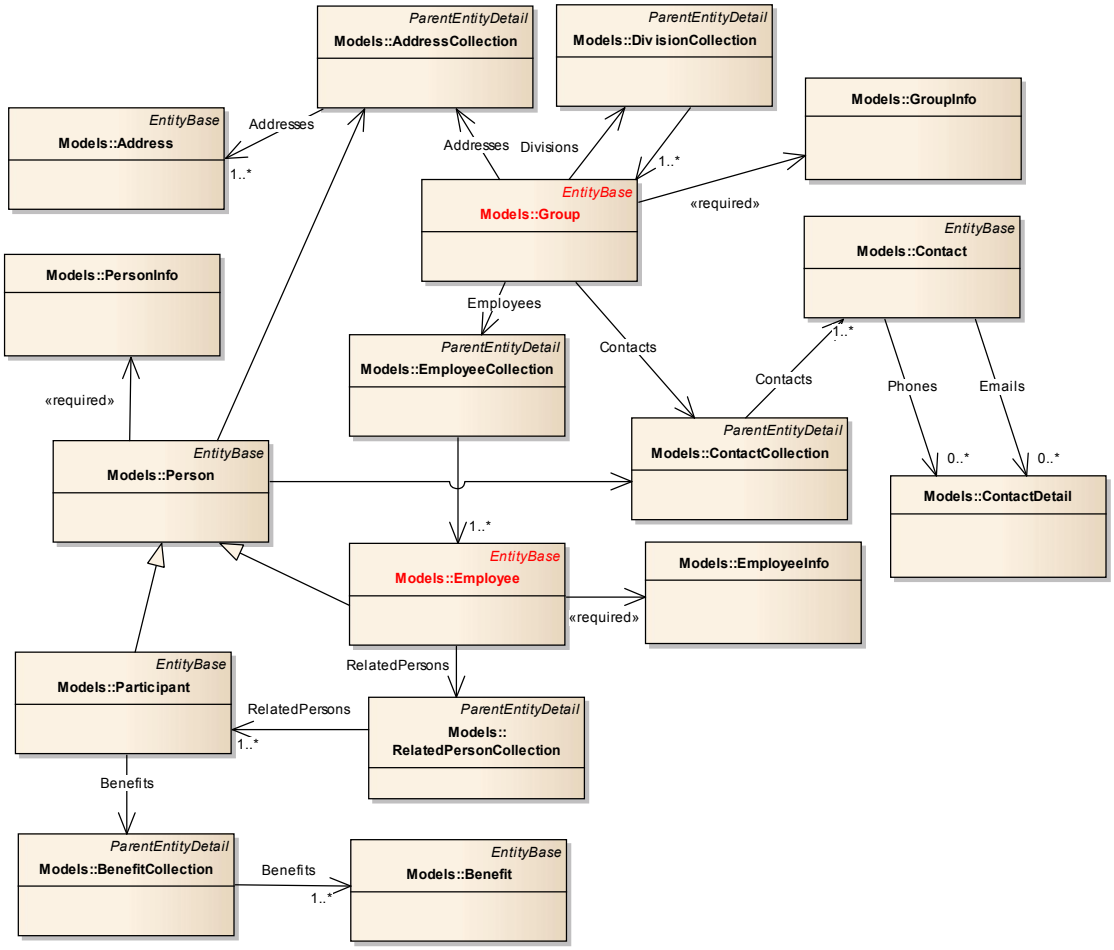

Figure 6. Canonical model's main entities: the payload of the data notifications

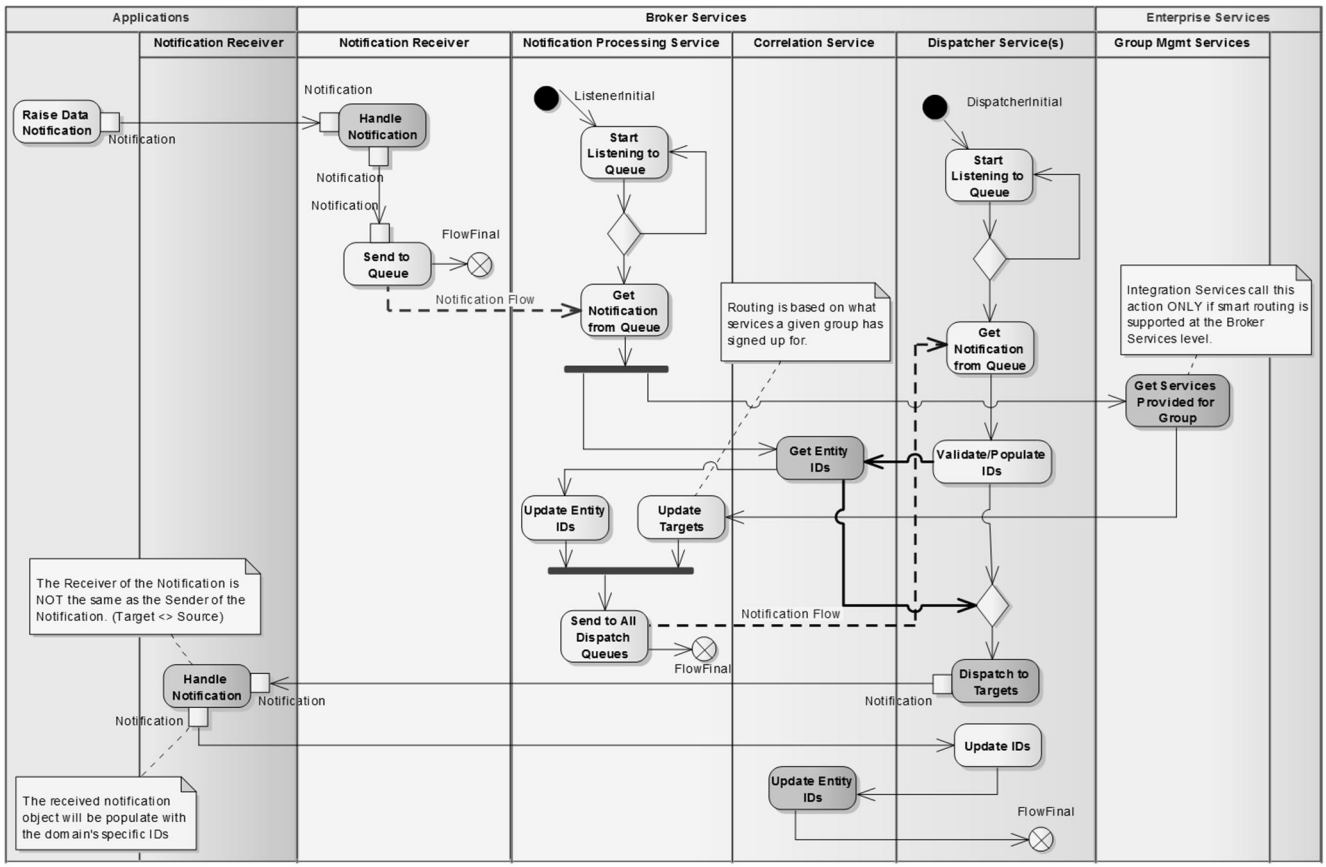

Figure 7. Enterprise integration activity model

V. A PHASED DELIVERY VIA PROTOTYPING

A. The Enterprise Integration Pilot Release

All of the artifacts presented so far are describing the asynchronous enterprise integration solution proposed and adopted by the client. It is important to note however, that this design was preceded by the implementation of a synchronous services middleware prototype, smaller in footprint and scope than the original design. This prototype was exposed only to a small set of customers, mainly in order to gain a deeper understanding of the data integration needs, the customer traction and adoption that such integration would yield, and the overall value it would bring to the Business.

Although early on it became evident that without such domain integration, independent systems would be forced to duplicate data and functionality, leading to potentially hazardous and undesired side effects as well as duplication of effort, a pilot version of the data integration was requested by the client, was implemented, and successfully released to the market.

Aside from the business value mentioned above, the prototype also allowed the teams to work out the means to a successful collaboration, to get familiar with each other's development processes and expectations, and to support each other during the system integration testing phase.

B. A Synchronous SOA-Based Prototype

For a faster turnaround, a combination of functional integration and data integration using synchronous services built around Microsoft's Windows Communication Framework (WCF) was designed and implemented as the pilot release. This prototype enabled two distinct business domains (benefits and payroll) and three isolated enterprise applications (one very large benefits application, a legacy payroll application, and a newly released, smaller payroll system) to share data common across multiple customers that these systems were actively servicing.

These customers for which common data required sharing and synchronization across systems, are provisioned via a lightweight web interface through which administrators have the ability to enable or disable the main integration facilities provided by the prototype – features that are specific to one or the other of the two domains. Since the initial set of customers to which this integration product was released was rather small (up to 50), a semi-manual provisioning activity was deemed acceptable. The web tool – developed as a Single Page Application (SPA) using Angular, Bootstrap, and JavaScript – just to mention a few of the technologies employed - also allowed power users to settle any customer identity (or reference) clashes that could not be automatically resolved via logical key matching.

Following a pure SOA approach and employing the industry-recommended SOA design patterns [9] [13], an integration middleware layer was implemented as part of this prototype solution, which included the correlation service and integration feature activation service along with a small database used to persist the data required by these services.

The middleware's purpose is to enable access to integration correlation data and resolve access queries against the unifying customer reference tables. It is responsible for activating or deactivating integration features for the targeted customers, and it also serves as an operational service layer to the provisioning web application.

At a high level, the architecture of the prototype and the communication paths between the integrating systems and the integration middleware are shown in Figure 8.

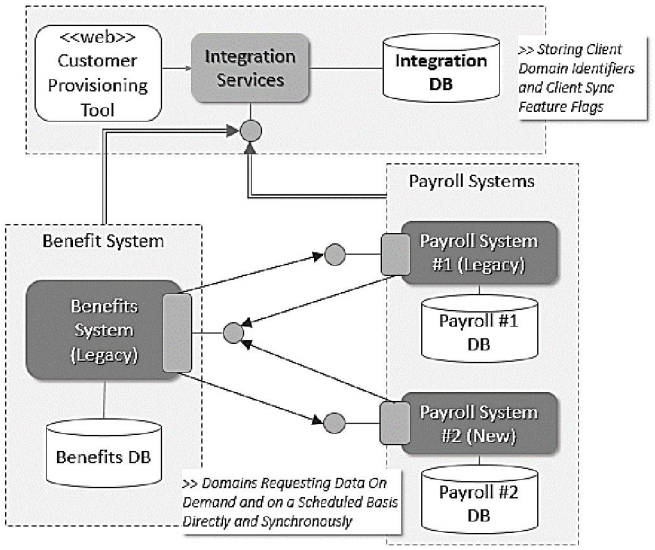

Figure 8. SOA-Based Synchronous Integration Prototype

The three integrating domains communicate directly with each other via service calls that encapsulate along with the payload, the correlation ID in order to reference a given customer. The ID is obtained from the Correlation Service (part of the integration services layer).

The service contracts designed for the domain-exposed endpoints (shown as horizontal lollipops in Figure 8) are simple yet symmetric (identical), allowing for a unified and consistent mechanism for requesting and exchanging data.

To overcome some of the architectural shortcomings of one of the legacy systems, changes to certain data required for synchronization had to be captured at the data layer (i.e., the database). For this purpose, a custom service component was designed using Microsoft's Change Tracking solution, which constitutes a lightweight implementation of the Data Change Tracking (DCT) solution.

This enabled monitoring and capturing the data updates from a standalone service component, without having to make changes to the domain services of the integrating system. Given the ease of implementation and ability to build it in isolation of other components, if was suggested as an alternate solution to the other integrating systems to compensate for the absence of reusable domain services, where and if applicable.

C. Key Benefits of Prototyping

Identifying specific areas of integration challenge and collecting valuable market insight from building and deploying a low-risk integration pilot (prototype) - even after high-level design effort and some middleware prototyping for the larger integration solution had already been wrapped by up the architecture team – were compelling enough arguments for the Product Management team.

Hence, the decision to spend the additional effort towards building a simplified synchronous functional integration pilot was made. As teams mobilized in the design elaboration and realization of this interim solution, several immediate benefits emerged, both for the development groups as well as for the decision-making entities.

Some of these benefits – relevant for this particular implementation – are captured below.

1) Refinement of the Integration Models and Contracts

Once concrete implementation artifacts started to take shape around the proposed models and interfaces, various gaps were identified and flagged with the design team. Such gaps included missing data fields for certain key domain entities – required for one system but not the others, ancillary lookup data mismatch across systems, and the stringent need for refining the composite logical (natural) keys used for uniquely identifying critical data entities (specifically, the aggregate roots) targeted for synchronization.

2) Identification of Edge-case synchronization issues

Certain customer data in one of the systems were found to have multiple representations in that system and such representations had to be handled accordingly by that system during the synchronous data exchange. This raised questions about handling data synchronization failures, both for the synchronous as well as the asynchronous implementation, which eventually lead to customizations to the durable message design realized by the service bus implementation, and the provisioning of nightly scheduled jobs that would retry sending or queuing failed notifications.

3) Defining Cross-Team Collaboration Processes

Multiple geographically dispersed teams were involved in the realization of the integration solution. Each system that would participate in the integration already had its own development team structure in place, its technical Subject Matter Experts (SMEs) and leads, its own practices and approaches to developing software.

Although all three teams involved in the original pilot implementation and delivery were following agile methodologies, the iteration schedules, task sizes and assignments, and even the way scrum meetings were run, differed quite a bit among them. Some effort was involved to iron out these differences and bring the teams to work together, to "speak the same language", to set the right expectations, and to meet the deliverables.

Moreover, issue escalation channels were established and the need to allow teams to *independently* test the integration points and, evidently, their own systems as they react to integration notifications became a critical item on everyone's list. This fact points us to the next benefit of prototyping – especially relevant in the case of systems' integration.

4) Building Synthetics

In general, when one thinks of software prototyping, perhaps throwaway and/or wireframe prototypes come to mind. Although some of the modules of the prototype presented here had to be modified or replaced in order to accommodate the larger integration solution, a big part of the middleware and components that encapsulated the production and consumption of notifications were fully reusable for the larger integration. However, in order to test those components while other teams were implementing their own handlers and dispatchers –without having to wait on everyone else to complete their implementation – specialized components that targeted the production of synthetic data and behavior – had to be built: both notification generators as well as mock notification handlers.

The idea behind the need for these synthetics is the distinction between unit testing and system integration testing; whereas the latter should not be allowed to proceed before independently validating first the integrating systems, the new components and frameworks, in isolation from each other.

Not only did these additional components provide a consistent testing framework for all teams, but also these synthetics would continue to be used and enhanced as needed, to support all ongoing unit and integration testing needs, including regression testing. These components greatly helped developers in catching integration point failures early on, before system integration testing (SIT) would commence. It identified the type of information that had to be monitored and captured to facilitate troubleshooting integration bugs, and made the overall integration testing much less painful than it would have been otherwise.

Although such simulators and data generators do not have a place in the final product, they are an absolute necessity in developing systems that must interact with each other – whether these systems are developed by the same company or are involving third party components. It is imperative to relieve individual component and/or system testing from any external dependencies in order to ensure proper validation of the system being built. Arriving at situations where it is unclear what the origin of the failure is or, worst, slowing down the development of your own system because of a faulty or unavailable external system, should always be avoided.

5) Aiding the Quality Assurance (QA) Teams with the Gradual Development of Integration Test Cases

Familiarizing themselves with a simpler system, the synchronous service-based prototype, gave the QA team ample time to prepare for the larger integration solution, identifying gradually the appropriate tests to be developed. This lead to a better comprehension of the key features of the integration that were mission-critical from an overall system perspective.

Finally, just as was the case for the development teams, multiple testing teams were assembled, facing similar challenges. Although with some additional effort and time, the QA teams successfully identified and implemented the necessary processes towards coordinating their testing efforts, preparing the test data, and collaborating effectively in order to validate the entire enterprise integration solution.

VI. CONCLUSION

Depending on the scope of system integration as well as the functional and non-functional requirements, creating the right frameworks and sub-systems that allow isolated domain systems to seamlessly share key enterprise data between them is a challenging undertaking. A variety of technology choices, architectural and modeling approaches and patterns exist, but features and limitations of the integrating systems, along with organizational, budgetary, technical, and technological constraints can make the integration task even more difficult. Generally speaking, in multi-domain enterprise systems, data integration and synchronization can be achieved in various ways. One of them – as the one presented here and in [1] – involves custom integration frameworks and components, using various enterprise integration patterns.

This paper presented an actual industry integration solution, explained via several structural and behavioral system models, and provided details on how the "maintain data copies" data integration pattern would be realized via a broker-based messaging middleware. The data exchanged between the various domains is encapsulated by the canonical model, which is the common data abstraction across the enterprise. This in turn is wrapped inside a context-based, generic, and reusable notification model, allowing systems to react to these notifications based on their own business rules.

This paper also captured essential enterprise integration patterns chosen for this solution and how they were employed, as well as the architectural topology designed to address specific functional and non-functional requirements. Central to the solution proposes here, the paper presented the common integration model and described how this model played the role of the semantic glue that unified the data exchange mechanism between the various integrating systems and components.

Following industry-recommended patterns and practices – yet custom-tailored to meet the specific client integration needs, the resulting architecture features scalability, extensibility, and high-availability – to mention just a few quality attributes. Concerning performance, it supports near-real-time data synchronization between systems and allowing them to operate without awareness of each other, while using their individual data formats, features, and domain rules.

Finally, the paper introduced a generalized integration prototype that was released to a reduced customer base as a pilot implementation, in order to test the market response to the new features enabled via integration. The prototype development proved valuable in several ways, as discussed in this article. The development of synthetics, in order to facilitate the imperative unit testing of all systems and components as a prerequisite to system integration testing, proved to be an invaluable byproduct of prototyping system integration.

A. Future Work

One of the benefits of being in the consulting business is the exposure to a diverse array of problems and challenges, leading the way by designing custom solutions, releasing the product to the market, and then moving on to new problems waiting to be solved. For the author of this paper, the solution

presented here has been a goal in itself, and it has been accompanied by a successful release to the market of the initial prototype, as well as the client's adoption of the extended asynchronous integration solution shown here. The responsibility of maintaining the integration middleware, as well as enhancing existing systems while adding new domains (such as Human Resource (HR) vertical(s) and Time and Attendance applications) into the integration mix stayed with the client for which the solution presented here was prototyped and delivered.

REFERENCES

[1] M. Iridon, "Enterprise Integration Modeling - A Practical Enterprise Data Integration and Synchronization Solution," FASSI 2015 : The First International Conference on Fundamentals and Advances in Software Systems Integration, pp. 23-30, August, 2015.

[2] Microsoft BizTalk Integration Platform. [Online]. Available from: https://www.microsoft.com/en-us/server-cloud/products/biztalk/ [retrieved: June, 2016]

[3] TIBCO Integration Platform. [Online]. Available from: http://www.tibco.com/products/integration [retrieved: May 2016]

[4] Microsoft, Data Integration. [Online]. Available from: https://msdn.microsoft.com/en-us/library/ff647273.aspx [retrieved: June, 2016]

[5] Microsoft, Integration Patterns. [Online]. Available from: https://msdn.microsoft.com/en-us/library/ff647309.aspx [retrieved: June, 2016]

[6] M. Iridon, "Technical Design Specifications for Enterprise Integration Solution," 2015, unpublished/internal document.

[7] G. Hohpe and B. Woolf, "Enterprise Integration Patterns; Designing, Building, and Deploying Messaging Solutions," Addison-Wesley, 2012.

[8] Microsoft, Functional Integration. [Online]. Available from: https://msdn.microsoft.com/en-us/library/ff649730.aspx [retrieved: June, 2016]

[9] T. Erl, "SOA Design Patterns," Prentice Hall, 2009.

[10] T. Erl et al., "Next Generation SOA: A Concise Introduction to Service Technology & Service-Orientation," Prentice Hall, 2014.

[11] M. Fowler, "Patterns of Enterprise Application Architecture," Addison-Wesley Professional, 2002.

[12] T. Erl, "Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services," Prentice Hall, 2004.

[13] T. Erl, "Service-Oriented Architecture (SOA): Concepts, Technology, and Design," Prentice Hall, 2005.

[14] Microsoft, Service Bus for Windows Server Quotas. [Online]. Available from: https://msdn.microsoft.com/en-us/library/dn441429.aspx [retrieved: June, 2016]

[15] M. Fowler, Anemic Domain Model. [Online]. Available from: http://www.martinfowler.com/bliki/AnemicDomainModel.html [retrieved: June, 2016]