An Interactive Pattern Story on Designing the Architecture of Clotho

ERNST OBERORTNER, Department of Electrical and Computer Engineering, Boston University, USA, ernstl@bu.edu
DOUGLAS DENSMORE, Department of Electrical and Computer Engineering, Boston University, USA, dougd@bu.edu
J. CHRISTOPHER ANDERSON, University of California, Berkeley, jcanderson@berkeley.edu

Clotho is a software platform tailored for the synthetic biology domain and it offers functionalities to develop domain-specific "Apps" to ease engineering novel biological systems. Clotho's current version — Clotho 2.0 — has several shortcomings regarding its architecture and the extensibility of its data model. In this paper, we report on the design process of architecting Clotho's new version — Clotho 3.0 — tackling the drawbacks of Clotho 2.0 and newly emerging quality and functional requirements. We tell an interactive pattern story that reflects our design decisions and chosen pattern-based solutions. To illustrate the architecture of Clotho 3.0 we exemplify a Data-Flow View through the architecture's components using a typical design activity of synthetic biologists.

Categories and Subject Descriptors: D.2.11 [**Software Architectures**] Patterns

General Terms: Design

Additional Key Words and Phrases: Software Architecture, Patterns, Architectural Design Decisions, Interactive Pattern Story, Synthetic Biology

1. INTRODUCTION

Synthetic biologists engineer complex and novel living systems to, for example, discover new drugs, cure diseases, or to the study the origin of life [Benner and Sismour 2005]. Designing, constructing, and synthesizing novel and complex genetic systems composed of existing parts of deoxyribonucleic acid (DNA) sequences with a well-known behavior is one of the major challenges of this emerging field [Andrianantoandro et al. 2006]. Computer-aided Design (CAD) tools have gained momentum to support synthetic biologists. But, such tools need to deal with large amounts of biological data, such as compositions of long DNA sequences and related meta-data information, for example, the fabrication lab, literature information, assembly protocols, or bio-safety features. Due to bio-design automation and fabrication methods, it is challenging in the synthetic biology domain to develop user-friendly CAD tools that keep up with the tremendous increase of biological data.

Clotho is a software platform [Densmore et al. 2009] especially tailored to manage large amounts of biological data. A Clotho Core provides a rich repertoire of interfaces to access databases that store biological data according

Authors' Address: E. Oberortner, 8 Saint Mary's Street, Boston, MA, 02215, email: ernstl@bu.edu
D. Densmore, 8 Saint Mary's Street, Boston, MA, 02215, email: dougd@bu.edu
J. C. Anderson, 512E Energy Biosciences Building, Berkeley, CA, 94720, email: jcanderson@berkeley.edu

to a pre-defined data model. Software developers then extend the Clotho Core functionalities with design-, lab-, and user-specific "Apps" making it easier for biologists to employ the biological data. Currently, various Apps exist to import, view, modify, assemble, create, and export novel DNA sequences, supporting biologists to design novel genetic systems.

In Clotho's current version — Clotho 2.0 — the Core and all user-specific Apps must reside on the users local machine, leading to the following shortcomings: (1) controlling the users' software environment is difficult, (2) versioning and updating each user's local Clotho installation is complex, and (3) predicting and managing data inconsistencies among several databases is hard. For App developers it is tedious to extend and customize the pre-defined Clotho 2.0 data model for App-, user-, or lab-specific requirements. Also, in Clotho 2.0 the integration of new Apps into an existing Clotho installation is time-consuming.

In this paper, we tell an interactive pattern story [Siddle 2011] on designing a new architecture for Clotho which tackles these shortcomings. The story is a vehicle to capture the taken architectural design decisions and chosen pattern-based solutions. Our story's patterns mainly stem from the POSA series [Buschmann et al. 1996; Schmidt et al. 2000; Kircher and Jain 2004; Buschmann et al. 2007a; 2007b]. We then illustrate a Data Flow View [Avgeriou and Zdun 2005] through the Clotho 3.0 architecture using a typical design activity of synthetic biologists. However, we do not present any technical details about the architectures implementation, the data exchange, and the data storage.

This paper is structured as follows: The next section, Section 2, gives basic background of the synthetic biology domain. In Section 3, we present the Clotho 2.0 architecture, its shortcomings, and our vision and requirements of the Clotho 3.0 architecture. In Section 4 we tell the interactive pattern story about the architectures design process and how the architecture tackles its requirements. Then, in Section 5 we present a data flow view through the core components of the Clotho 3.0 architecture. We conclude and list some future work in Section 6.

2. BACKGROUND: WHAT IS SYNTHETIC BIOLOGY?

DNA serves as the storage molecule of biological information for the construction and function of cells. Synthetic biologists try to construct new DNA strands which produce proteins that behave in a predictable way by expressing genes in a controlled way. DNA is interpreted to build a protein through a two-step process: transcription and translation. Transcription is the process during which mRNA (messenger ribonucleic acid) is transcribed from DNA by RNA-polymerase. Translation is the process of translating the transcribed mRNA to a protein, by using a ribosome molecule according to the genetic code [Trun and Trempy 2003].

DNA transcription is initiated when a RNA polymerase (RNAP) molecule recognizes and binds to a promoter sequence. The DNA transcription ends when the RNAP transcribes the terminator sequence. The binding of RNAP to the promoter sequence can either be enhanced or repressed by Transcription Factors (TFs) and small molecules. The ribosome recognizes and binds to a Ribosome Binding Site (RBS) which is an untranslated sequence at the beginning of the mRNA transcript and translates the mRNA into protein.

The transcription process includes various functioning and well-engineered genetic parts. A promoter initiates the DNA transcription and a terminator terminates the DNA transcription. TFs enhance or repress promoters, and an RBS is an untranslated sequence important for the translation process. Such parts are of particular importance in synthetic biology in order to construct devices that behave in a controlled and predicted way.

To enhance the understandability of synthetic biology, we draw an analogy between logic circuits and genetic devices in Figure 1. A NOR gate has two input signals and one output signal that depends on the two input signals. The output signal is "on" only if both input signals are "off" (see the truth table in Figure 1). In an equivalent way it is possible to control — turn "on" or "off" — the expression of genes. As illustrated in Figure 1, a genetic NOR gate [Tamsir et al. 2011] is a genetic device composed of several genetic parts. It starts with two inducible promoters, which initiate — dependent on environmental input molecules — an expression of a reporting gene, such as a green fluorescent protein (GFP). Only if both inducible promoters are "off" (i.e. the environmental molecules do not bind with them), the repressor allows a downstream promoter to initiate the reporter. If both input signals are "on" or just

Fig. 1: Logic and Genetic NOR Gates

one of them, the downstream promoter is repressed and hence, the reporter will not be expressed. Similarly, various other genetic devices, such as AND, OR, or NAND, can be constructed to control the gene expression.

In a top down workflow, synthetic biologists specify the design of a genetic circuit. First, the designer specifies the functionally of the circuit, for example, to fluoresce yellow in the present of a small molecule. The design specification is then "compiled" into a series of motifs (a genetic regulatory network). These motifs are assigned to biological parts (promoters, ribosome binding sites, genes, terminators, etc). Physical samples and optimal assembly instructions for these samples are then sent to liquid handling robots. Finally the results of the design are then saved back to repositories of biological designs so that they (and their characterization data) can be used in future design.

Current design automation workflows in the synthetic biology domain include bio-specific tools (e.g. DNA sequence manipulation programs, such as A plasmid Editor (ApE)[1] or Vector NTI), webtools (e.g. Genedesign [4] for codon optimization), and community shared database resources (e.g. the Registry of Standard Biological Parts and Genbank). For example, a synthetic biologist might store the map of a plasmid in a Genbank file generated from ApE on her hard drive, document its composition as a list of BioBrick parts in a Google Doc, maintain lists of stocks containing the plasmid in an Excel file, and describe the literature references on a wiki. Resultant, a variety of loosely-coupled and isolated tools exists and the bulk of data is managed through ad hoc tools.

3. REQUIREMENTS ON CLOTHO 3.0

In this section we explain the shortcomings of Clotho's current architecture — Clotho 2.0 — and our vision and requirements on Clotho 3.0.

Clotho users are individuals, such as synthetic biologists, who utilize various Clotho Apps to design novel biological systems. The Clotho user utilizes the Apps using a Clotho client, such as a web browser or a stand-alone application. A Clotho client is a piece of software that host pre-defined and user-specified set of Clotho Apps usable by Clotho users.

3.1 Clotho 2.0 and its Shortcomings

The Clotho Core builds the interface between the Apps and a Clotho database and is responsible to retrieve and store the Apps' required data. From a pattern perspective, the Clotho 2.0 architecture (see Figure 2) recalls the REPOSITORY architectural patterns [Shaw and Garlan 1996].

[1] http://www.biology.utah.edu/jorgensen/wayned/ape/

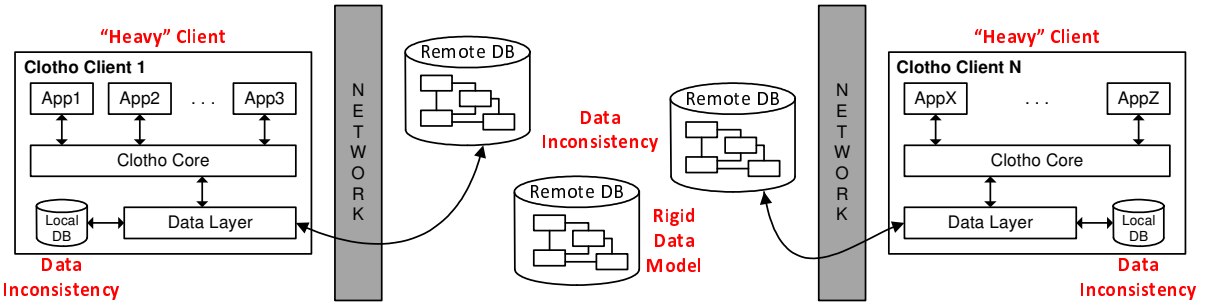

Fig. 2: The Clotho 2.0 Architecture

In Clotho 2.0, we have identified the following shortcomings:

Shortcoming 1: "Heavy" Clients.
Currently, the Clotho Core and all the user-required Apps are installed on a user's local machine, i.e. on one client. Such heavy clients complicate the automation of compiling, packaging, installing, and updating process of Clotho 2.0 distributions. For example, the biologists in our lab utilize a different set of Clotho Apps then our collaborating biologists. Currently, manual efforts are needed to compile, package, and install Clotho distributions. Moreover, in case of software updates of the Apps or the Clotho core, manual efforts are needed to update existent Clotho distributions.

Shortcoming 2: Data Inconsistencies.
As shown in Figure 2 every Clotho Core can access a local database. The user decides at Clotho's startup either to connect to the local or to a remote database. Clotho 2.0 does not incorporate synchronization mechanisms to keep the local data consistent with the remote data. There is no way to push the local data into a remote database. And there is no pull mechanism to fetch data from a remote database into the local database. Furthermore, Clotho 2.0 does not support yet to keep the data amongst the remote databases consistent. For example, if a Clotho user utilizes, for example, an App to fetch data from the connected remote database and the requested data is not available, then an empty data set will be returned. However, there is a chance that the requested data is stored in a different remote database. The loose coupling between the local and the remote databases, as well as among remote databases, can result in serious data inconsistencies.

Shortcoming 3: Rigid Data Model.
In Clotho 2.0 we offer a rich data model (i.e. a database schema) to store various types of biological data into the databases, such as composite parts of DNA sequences, their basic parts, physical samples, assembly formats, and meta-information about the fabrication lab or bio-safety features. Current Clotho Apps adhere to the data model. It is tedious for App developers to extend and/or customize the Clotho 2.0 data model. For example, develop a new Clotho App focusing on biological aspects not captured by the Clotho 2.0 data model, App developer must squeeze the data somehow into the Clotho 2.0 data model, utilizing various data model features and relationships in a wrong way.

3.2 Our Vision of Clotho 3.0

In Figure 3 we give a visionary perspective of Clotho 3.0, physically separating the clients, the Clotho Core, and the database. Since clients host the Apps of Clotho users we envision to support various types of clients, such as web-based, lightweight, and mobile clients as well as — to maintain backward compatibility — Clotho 2.0 clients. The Apps communicate with the Clotho core over the client. A Clotho Core services the requests and, eventually, executes the behavior of the Apps. Therefore, every Clotho Core connects to at least one database. Clotho Cores and databases do not necessarily need to be physically separated. In our vision, Clotho Cores can also communicate with other known Clotho Cores, for example, to ask for some biological data that is not present in one Clotho Core's databases.

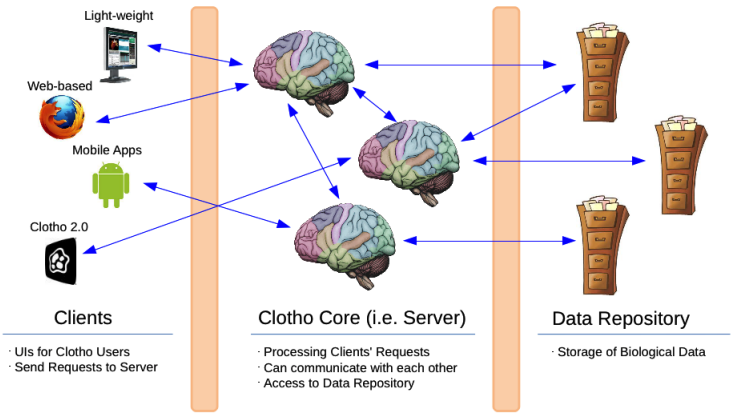

Fig. 3: Our Vision of Clotho 3.0

To make our vision come true, we need to address the following requirements that overcome the Clotho 2.0 shortcomings and to drive the quality and success of Clotho 3.0 and its architecture.

Requirement QR-1: Decoupling the Apps from the Clotho Core.
To overcome Shortcoming 1 (see Section 3.1), we envision a physical separation of the Clotho Core and the Apps, making it also easier to test and debug them separately. A physical separation, however, requires a communication solution, which should be transparent to the Clotho user and it should not influence the performance of the Apps and their hosting client. We believe that decoupling the Apps from the Clotho Core will overcome the current manual distribution process.

Requirement QR-2: APIs for Efficient Data Access.
In the design automation process of novel and complex biological systems, common data and work flows exist where data retrieval, data modification, and data storage are the main subjects. For example, biologists utilize Apps to (1) query basic parts of DNA sequences with specific behaviors and characteristics from the database, (2) execute algorithms on the loaded parts's data to calculate optimized assembly strategies, (3) control liquid handling robotics that automatically assemble the parts physically into systems according to the algorithms' results, and (4) measure and analyze the biological system's dynamic behavior. Clotho 3.0 must support database APIs to the App developers that efficiently access and persist existent and novel biological data. However, the data retrieval does not have serious time-critical constraints. Data needs to be retrieved in a user-accepted time range making the user not wait long until the data arrives.

Requirement QR-3: Seamless Integration of Clotho 2.0 Clients.
To achieve backward compatibility, we require that Clotho 2.0 clients can integrated into the Clotho 3.0 architecture and/or infrastructure. The reason is that Clotho 2.0 is currently being used by various biologists and architectural modifications to achieve our vision are inevitable.

Requirement QR-4: Customizable Data Model.
Biologists permanently discover proofs of currently unknown biological behavior. Defining a data model that captures all known aspects of biology is, for time being, virtually impossible. According to Shortcoming 3 (see Section 3.1) we require some mechanisms to hook novel data models into Clotho 3.0 and to utilize or share existing ones. For example, if a new App incorporates biological knowledge that is not captured in an existent data model, then, the App developer should be able to specify the new App's data model. The novel data models and Apps can then also be shared to other biologists, App developers, and the community.

4. AN INTERACTIVE PATTERN STORY ON THE ARCHITECTURAL DESIGN PROCESS

In this section, the paper's main contribution, we tell an interactive pattern story about the design process of Clotho's new architecture and its data model. The story reflects the taken design decisions in order to fulfill the aforementioned requirements. Every pattern-based solution to one design decision raises additional requirements that we also incorporate into various steps of the pattern story. In Figure 4 we orient the reader through the pattern story.

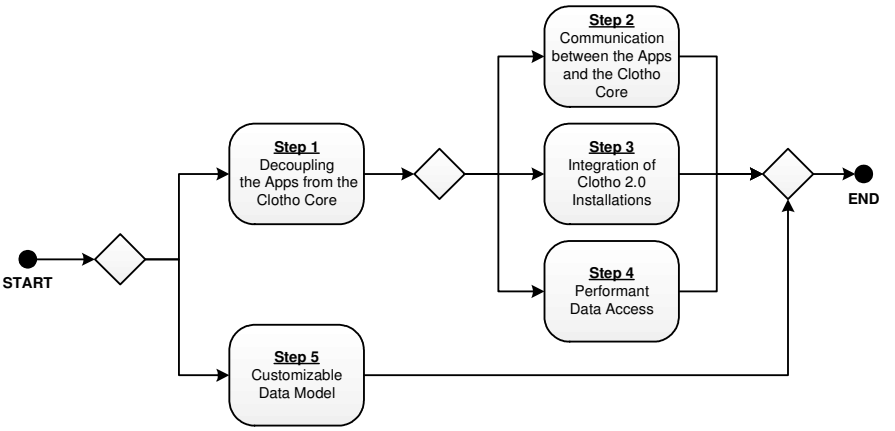

Fig. 4: Map of the Interactive Pattern Story

Start

> ***Option 1:*** *Start with* ***Step 1*** *if you wish to discover the Clotho 3.0 architecture.*
> ***Option 2:*** *Start with* ***Step 5*** *if you wish to discover the Clotho 3.0 data model.*

Step 1: *Decoupling the Apps from the Clotho Core (QR-1).*

In Clotho 2.0, the Apps and the Clotho Core are tightly coupled and installed together on a Clotho user's local machine. Since we had already a separation into an application logic (i.e. the Apps) and a data access logic (i.e. the Clotho Core) we thought about separating the Apps and the Clotho Core physically. Hence, we decided in favor of a CLIENT/SERVER [Avgeriou and Zdun 2005] architectural style. The separation into a client and a server component improves the testing and debugging facilities, since both components can be tested and debugged separately.

We started defining the duties and responsibilities of the Clotho core as a server. We agreed on following a component-based architecture, dividing the components into three inter-related groups: (1) handle to communication between the clients and the Clotho core, (2) do the processing of the clients' requests, and (3) access the data repositories. As a result (see Figure 5), the new Clotho Core resembles the LAYERS pattern [Buschmann et al. 1996] consisting of a **Communication Layer**, an **Execution Layer**, and a **Data Access Layer**.

The data exchange between client and server follows the COMMAND PROCESSOR pattern [Buschmann et al. 1996] that builds on the COMMAND pattern [Gamma et al. 1994]. In the Request Handling layer we integrated an Executor component, which receives the incoming request objects from the Listener component of the Communication Layer. The Executor unwraps the requests' data objects and instantiates an Assistant following the FACTORY METHOD pattern [Gamma et al. 1994] that perform the actual processing of the data objects. An Assistant calls the Data Access Layer to access the data stored in the data repositories. To return the result to the requesting client, the Assistant invokes the assigned Callback-Handler.

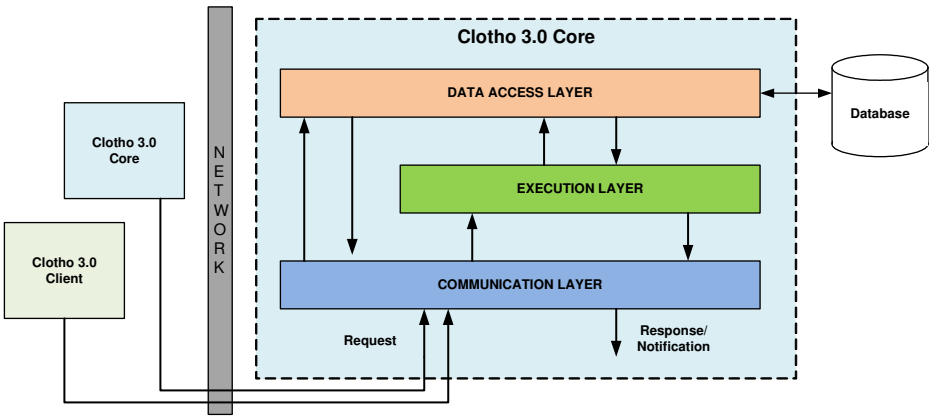

Fig. 5: The LAYERS of the Clotho 3.0 Core

Option 1: If you wish to discover the design process of the Core's communication layer, then turn to **Step 2**.
Option 2: If you are interested in the design of the data access layer, then go to **Step 3**.
Option 3: If you want to see the integration of existing Clotho 2.0 installations into the new Clotho architecture, then move to **Step 4**.

Step 2: *Communication between the Apps and the Clotho Core.*

In Clotho's new architecture, the Clotho Core and the clients that host Apps are loosely coupled and physically separated. The Apps communicate with the Clotho Core to perform various actions, such as querying, modifying, or storing biological data that resides in a database. Following a CLIENT/SERVER architectural style raises several additional requirements on the Clotho 3.0 architecture, especially on the Clotho Core:

Scalability: The Clotho Core services various requests from the clients and other Clotho Cores. In case of a huge load, i.e. a lot of requests arrive simultaneously, a Clotho Core must scale with the payload and should not influence the performance and work of the requesting clients or Clotho Cores.

Supporting various Clients and Requests: Because of the physical separation of the Apps and the Core, clients need to communicate with the Core over a network using a common communication protocol. The Core should be, however, agnostic to the client's platform and implementation in order to support lightweight, web-based, mobile, as well as Clotho 2.0 clients. Apps, on the other hand, can send various types or requests to the Core, such as to query data from the database or to execute deployed functions. The Core needs to understand the request and process it appropriately.

Session Handling: Similar to designing software architectures, novel biological systems are not fully designed within one single design session. Therefore, we need to keep track of the clients and Apps' state. Also, to incorporate , we require session handling. A suitable session handling solution could also handle user authentication, privacy, and security mechanisms.

Asynchronous Communication: Whenever a user utilizes a Clotho App sends a request to the Core, the client should not stall. For example, the verification of a designed synthetic biological system against bio-safety and biophysical requirements, can take a long time. In such a case, Clotho users should be able to continue their work while they are waiting for the verification results. Asynchronous data exchange between the Clotho clients and the Clotho core is inevitable.

Notification Mechanisms: In Clotho 3.0 we envision to update the Clotho users automatically in case of software and data updates. Therefore, the Core needs to push updates specific to the users' interests to clients and Apps. For example, if a biologist is interested in the arrival of new parts of DNA sequences in the database, then the Core should notify them accordingly. Also, if a user initiates a long-running algorithm on the Core, then the Core needs to let the user know when the algorithm's execution has been finished. The requirement of notification mechanisms is closely related to the requirement of asynchronous communication.

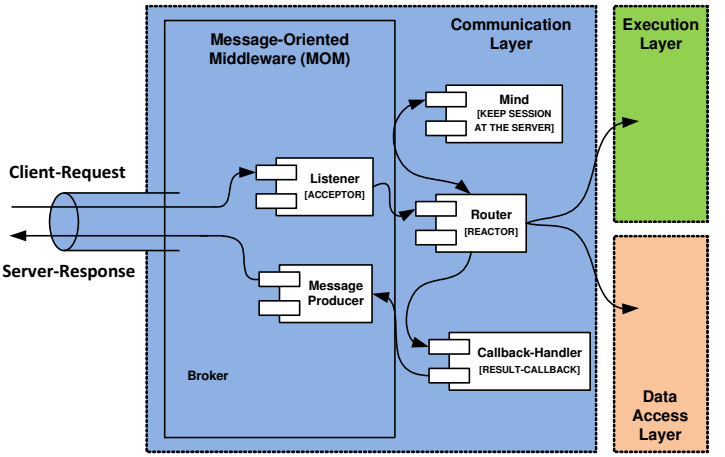

Fig. 6: The Communication Layer of the Clotho 3.0 Core

In Figure 6 we present the architecture of the Core's communication layer. The communication layer receives requests from the clients and other Cores and responds to or notifies clients and other Cores accordingly. Clotho's communication layer consists of three inter-related components — Router, Mind, and Callback-Handler — that are built atop a message-oriented middleware (MOM). In our MOM, a BROKER offers communication channels to the Clotho clients on which messages are being exchanged which encapsulate the request data. On every communication channel, a channel listener waits for incoming messages and forwards them to the Router component. This channel listener component recalls (1) the SERVER REQUEST HANDLER pattern [Voelter et al. 2004] since it deals with the incoming messages received over the network and (2) the ACCEPTOR component of the ACCEPTOR-CONNECTOR pattern [Buschmann et al. 2007a] since it accepts incoming requests. The router router evaluates the data in the request and forwards the request accordingly. For example, in case of a database query request, the Router forwards the request to the data access layer. In case of an execution request, the Router forwards the request to the Execution layer. The router component recalls the following patterns: (1) the INVOKER pattern [Voelter et al. 2004] since it deals with the requests and how they are dispatched, (2) the REACTOR pattern [Schmidt et al. 2000] since it demultiplexes and dispatches the incoming requests.

Utilizing a MOM enables that various types of clients can communicate with the Core on a common communication protocol. Also, the router component is technology agnostic, making it possible to switch to another communication middleware quite easily. To support asynchronous data exchange, we decided in favor of the RESULT-CALLBACK pattern [Voelter et al. 2004]. A Callback-Handler returns the Clotho Core's response to the requesting clients. The Router initiates for every incoming request an appropriate Callback-Handler which ultimately responds to the requesting clients.

To achieve session handling, the KEEP SESSION AT THE SERVER session-handling pattern got our attention [Sorensen 2002]. There, we integrated a Mind component into the communication layer that stores data updates and user's sessions and, hence, can notify Clotho users appropriately. However, the concrete design and implementation

of the mind component is still work in progress. To notify users and clients about relevant data changes, we foresee to follow the PUBLISHER-SUBSCRIBER pattern [Buschmann et al. 1996]. First, every Clotho user can subscribe to topics they want to be notified about. The subscription information is stored in the database and the Communicator component utilizes this information to notify the users appropriately. However, the integration of the PUBLISHER-SUBSCRIBER pattern into the Clotho 3.0 architecture is currently work in progress.

> **Option 1:** Keep on reading with **Step 3** to learn about the integration of Clotho 2.0 installations into the new architecture.
> **Option 2:** Move to **Step 4** if you are interested in the architecture of the data access layer of Clotho 3.0.
> **Option 3:** If you want to discover Clotho's new data model, then jump to **Step 5**.

Step 3: *Integration of Existent Clotho 2.0 Clients (QR-3).*

After having decided in favor of a CLIENT/SERVER architectural style and the design of the new Clotho Core's communication layer, one challenge we had to face was to integrate of existing Clotho 2.0 clients into the Clotho 3.0 infrastructure in order to achieve backward compatibility.

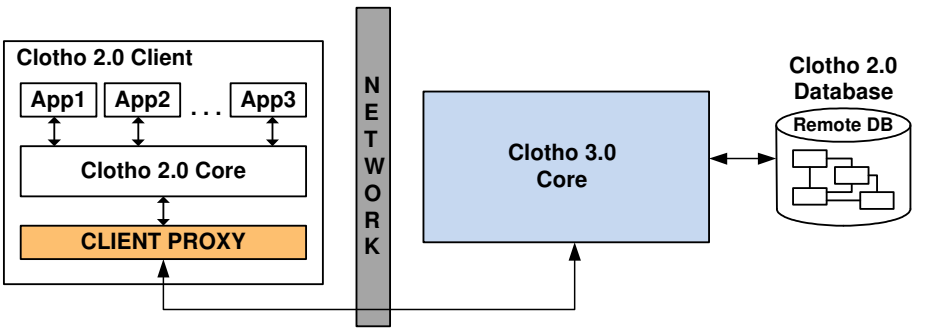

Fig. 7: Integrating Clotho 2.0 clients into the Clotho 3.0 infrastructure

In Figure 7 we illustrate the utilization of the CLIENT PROXY pattern that replaces the Clotho 2.0 database layer. The client proxy offers the same database access interfaces to the Clotho 2.0 Core, maps database access invocations to the communication protocol, and sends a database access request to a Clotho 3.0 Core. At the server-side, the Clotho 3.0 will not recognize that particular requests come from Clotho 2.0 clients. However, we need to migrate the Clotho 2.0 data model to the Clotho 3.0 data model and integrate existing Clotho 2.0 databases into the Clotho 3.0 infrastructure.

> **Option 1:** Keep on reading with **Step 4** to learn about the the architecture of the data access layer of Clotho 3.0.
> **Option 2:** If you are interested in the new Clotho data model, then move forward to **Step 5**.

Step 4: *APIs for Efficient Data Access (QR-2).*

The utilization of the LAYERS pattern provokes to dedicate one layer for the database access and persistence. In Figure 8 we illustrate the components of the Clotho 3.0 data access layer and their collaborating layers. Designated components of the communication and execution layer invoke the database API to perform the requested database operations, such as storing, querying, or deleting data. To enhance the efficiency of the database access, we integrate a cache component that keeps frequently used objects in-memory. The database API first contacts the cache and if the

requested data is not present in the cache, then the database API forwards the persistor which queries data from and persists data to a database.

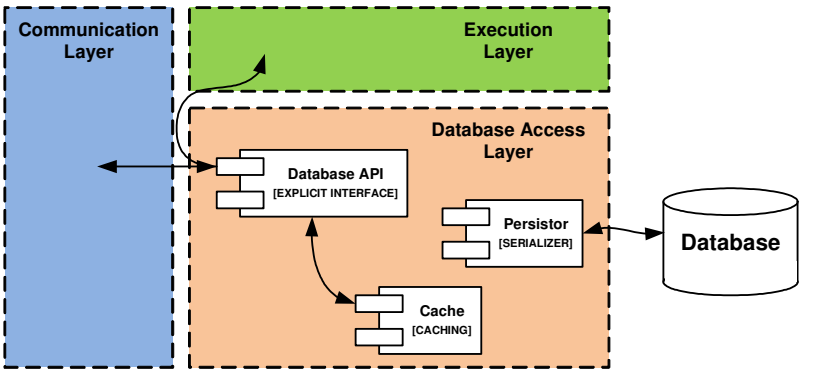

Fig. 8: The Data Access Layer of the Clotho 3.0 Core

The database API recalls the EXPLICIT INTERFACE [Buschmann et al. 2007a] because it hides the data access implementation from to collaborating layers and components. the components and hides its implementation The cache component follows the CACHING pattern [Kircher and Jain 2004] by storing data in a fast-access buffer to achieve a better performance to retrieve biological data. The persistor resembles the SERIALIZER pattern [Martin et al. 1997] since it stores and retrieves data from the databases.

> **Option 1:** If you want to discover our vision of the Clotho 3.0 data model, then keep on reading with **Step 5**.
> **Option 2:** Otherwise, you have reached the end of the story.

Step 5: *Customizable Data Model (QR-4).*

We provide in Figure 9 an excerpt of the Clotho 3.0 data model and the classes. The `ObjBase` class is the root class of the Clotho 3.0 data model and assigns to every object in Clotho 3.0 a unique identifier. In Clotho 3.0, we offer two possibilities how to specify customized and App-specific data models:

(1) App developers can specify schemas that consists of various fields that are WRAPPERS [Gamma et al. 1994] around low-level and database-related data types. To define references between schemas, we provide a `XREF` field type. Every schema can have multiple instances. For example, we can instantiate an 'Institution' schema consisting of a name string field with multiple instances such as 'UC Berkeley' or 'Boston University' which represent the instance name of the 'Institution' schema. For the specification of schemas, we utilize the JavaScript Object Notation (JSON).

(2) Clotho 2.0 offers a rich data model and we did not want to change it. Therefore, App developers can also specify classes and data models that inherit from the `ObjBase` class. This extension mechanism facilitated the integration of the Clotho 2.0 data model into the Clotho 3.0 infrastructure. We believe that this solution enables to hook novel data models into Clotho 3.0 easily.

Both approaches allow App developers to specify validation and indexing mechanisms to sustain the data in the databases clean and to enhance the performance of the database access. We foresee that App-specific data models will be shared between App developers and the synthetic biology community at large. For example, we have implemented the Clotho 2.0 data model using Clotho 3.0's schema-based data model definitions. Therefore, we introduced a `Sharable` interface.

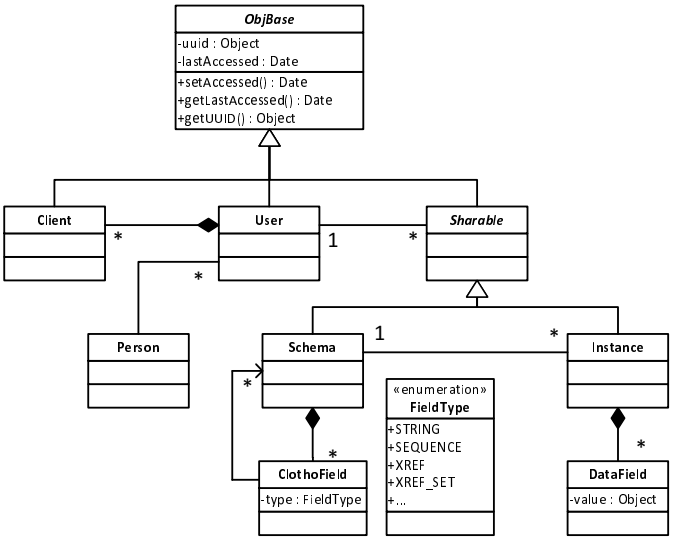

Fig. 9: The Clotho 3.0 Data Model

Option 1: If you started reading the interactive pattern story with Step 5, and if you are interested in the design of the new Clotho Core, then move back to **Step 1**.
Option 2: Otherwise, you have reached the end of the story.

The End

5. A DATA FLOW VIEW THROUGH THE CLOTHO CORE ARCHITECTURE

We exemplify in Figure 10 a data flow view [Avgeriou and Zdun 2005] through the Core's components of Clotho 3.0. We utilize a typical design activity of the synthetic biology domain. In this scenario, the Clotho user utilizes Eugene [Bilitchenko et al. 2011] to construct all valid compositions of parts of DNA sequences. Therefore, a web-based client hosts the EugeneScripter App and sends execution requests to the Clotho Core to execute the user-specified Eugene scripts based on the in the database residing parts of DNA sequences. When the execution of the Eugene script is being finished, the Clotho Core returns the resulting data — i.e. valid part compositions — to the requesting client. The client then forwards the incoming data to the EugeneScripter App, which displays the data suitably.

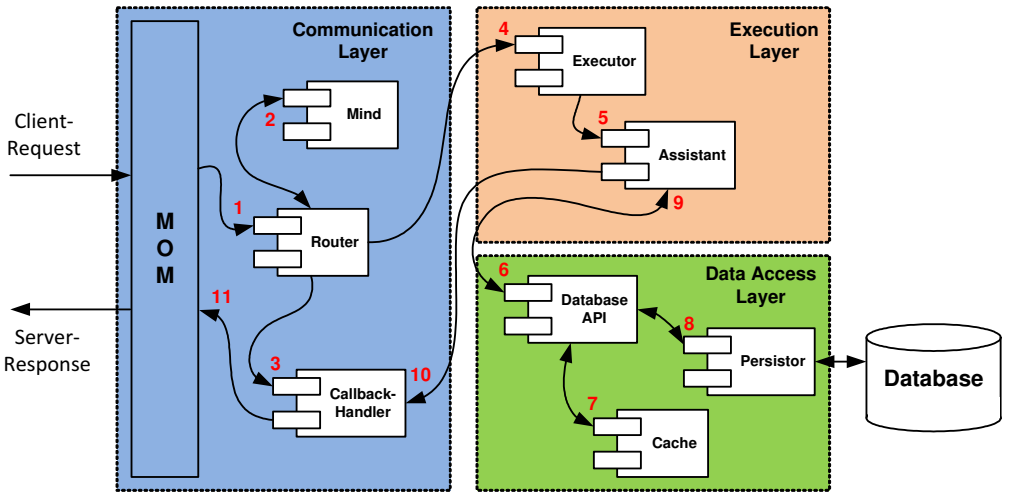

Fig. 10: A Data Flow View through the Clotho 3.0 Core

At the server-side, the data flows through the Core's components as follows:

(1) The Router receives the incoming client request from the underlying MOM.
(2) The Router contacts the Mind which creates a new session if there is no active session of the requesting client.
(3) The Router instantiates and assigns a Callback-Handler to the incoming request information in order to respond the requested data accordingly.
(4) Since the Router has received an execution request, the Router forwards the request's data to an appropriate Executor of the Execution Layer.
(5) The Executor instantiates an appropriate Assistant that executes the in the request packaged Eugene script.
(6) In order to request the data repositories, the Assistant invokes an appropriate method of the Database API.
(7) The Database API queries the requested data first from the cache. If the data is present in the cache, then the Database API returns the data to the Assistant.
(8) If the data is not present in the cache, then the Database API invokes the Persistor component which queries the data from the database.
(9) Then, the Database API returns the requested data to the Assistant.
(10) After the Assistant is being finished with the Eugene script execution, it invokes the to the request assigned Callback-Handler.
(11) Ultimately, the Callback-Handler utilizes the underlying MOM to return the resultant data to the client for eventual further processing.

6. SUMMARY AND FUTURE WORK

In this paper, we told an interactive pattern story about the design process of the Clotho 3.0 architecture. The new architecture follows a CLIENT/SERVER architectural style and tackles several drawbacks of the Clotho 2.0 architecture. The story reflects the design decision that we were facing in various design session in order to create an architecture that fulfills several quality and functional requirements. In the story, we align the architecture's layers and components with patterns from the pattern literature. We also visualized the architecture using a data flow view, based on a typical design activity of the synthetic biology domain.

In the future, we plan on integrating the the following requirements. We require management facilities and an App-store, making it possible that users can download and install Apps on demand. One other important aspect is to achieve privacy and security regarding biological data and the client-server communication. For example, we can enrich the Mind component and the underlying MOM for securing biological data from authorized access and the communication data exchanged between the clients and the Core. Also, we will further flesh-out the communication between multiple Clotho Cores. This is of particular interest from a synthetic biological standpoint because one Clotho Core can contact other Clotho Cores when some requested biological data is not available.

We believe that the Clotho platform, its offered functionalities, and extension mechanisms will ultimately lead to an eco-system of Apps that ease the engineering process of novel synthetic biological systems.

Acknowledgments

Special thanks go to our other team members, especially Swapnil Bhatia, Stephanie Paige and Maxwell Bates.

We gratefully thank our shepherd, Stefan Sobernig, and all participants of the PLoP writers' workshop — Tony Edgin, Eduardo Guerra, Kirstin Heidler, Michael John, Jiwon Kim, Nicco Kunzmann, Marko Leppanen, Juan Reza, Youngsu Son, Francois Trudel, Rebecca Wirfs-Brock and Joseph Yoder. All provided invaluable feedback on improving the manuscript.

We also thank Neil Harrison for performing a pattern-based architecture review (PBAR) of the architecture while having dinner at the PLoP 2012 conference.

This work is fully supported under NSF Grant 1147158.

REFERENCES

ANDRIANANTOANDRO, E., BASU, S., KARIG, D. K., AND WEISS, R. 2006. Synthetic biology: new engineering rules for an emerging discipline. *Molecular Systems Biology 2*.

AVGERIOU, P. AND ZDUN, U. 2005. Architectural Patterns Revisited — A Pattern Language. In *EuroPLoP*. 431–470.

BENNER, S. A. AND SISMOUR, A. M. 2005. Synthetic biology. *Nature Reviews Genetics 6,* 1.

BILITCHENKO, L., LIU, A., CHEUNG, S., WEEDING, E., XIA, B., LEGUIA, M., ANDERSON, J. C., AND DENSMORE, D. 2011. Eugene: A domain specific language for specifying and constraining synthetic biological parts, devices, and systems. *PLoS ONE 6,* 4, e18882.

BUSCHMANN, F., HENNEY, K., AND SCHMIDT, D. C. 2007a. *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*. Wiley.

BUSCHMANN, F., HENNEY, K., AND SCHMIDT, D. C. 2007b. *Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages*. Wiley.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. 1996. *Pattern-oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., New York, NY, USA.

DENSMORE, D., VAN DEVENDER, A., JOHNSON, M., AND SRITANYARATANA, N. 2009. A platform-based design environment for synthetic biological systems. In *The Fifth Richard Tapia Celebration of Diversity in Computing Conference: Intellect, Initiatives, Insight, and Innovations*. TAPIA '09. ACM, New York, NY, USA, 24–29.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software* 1 Ed. Addison-Wesley Professional.

KIRCHER, M. AND JAIN, P. 2004. *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*. Wiley, Chichester, UK.

MARQUARDT, K. 1999. Patterns for Plug-Ins. In *EuroPLoP*.

MARTIN, R. C., RIEHLE, D., AND BUSCHMANN, F., Eds. 1997. *Pattern Languages of Program Design 3*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

SCHMIDT, D., STAL, M., ROHNERT, H., AND BUSCHMANN, F. 2000. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Wiley.

SHAW, M. AND GARLAN, D. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall.

SIDDLE, J. 2011. "Choose Your Own Architecture" — Interactive Pattern Storytelling. *Transactions on Pattern Languages of Programming 2*, 16–33.

SORENSEN, K. E. 2002. Session Patterns. In *EuroPLoP*, A. O'Callaghan, J. Eckstein, and C. Schwanninger, Eds. UVK - Universitaetsverlag Konstanz, 301–322.

TAMSIR, A., TABOR, J. J., AND VOIGT, C. A. 2011. Robust multicellular computing using genetically encoded NOR gates and chemical 'wires'. *Nature 469,* 7329, 212–215.

TRUN, N. J. AND TREMPY, J. E. 2003. *Fundamental Bacterial Genetics*. Blackwell Science.

VOELTER, M., KIRCHER, M., AND ZDUN, U. 2004. *Remoting Patterns – Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*. Wiley Series in Software Design Patterns. J. Wiley & Sons, Hoboken, NJ, USA.

APPENDIX

A. THE UTILIZED PATTERNS IN THE CLOTHO 3.0 ARCHITECTURE

Pattern Name	**Problem Statement**	**The Pattern's Solution**
LAYERS [Buschmann et al. 1996]	Imagine that you are designing a system whose dominant characteristic is a mix of low- and high-level issues, where high-level operations rely on the lower-level ones.	Structure your system into an appropriate number of layers and place them on top of each other.
BROKER [Buschmann et al. 1996]	When distributed components communicate with each other, some means of inter-process communication is required. If components handle communication themselves, the resulting system faces several dependencies and limitations.	Clients access the functionality of servers by sending requests via a BROKER. A broker's tasks include locating the appropriate server, forwarding the request to the server and transmitting results and exceptions back to the client.
COMMAND PROCESSOR [Buschmann et al. 1996]	An application that includes a large set of features benefits from a well-structured solution for mapping its interface to its internal functionality. You often need to implement services that go beyond the core functionality of the system for the execution of user requests.	The Command Processor pattern builds on the COMMAND design pattern [Gamma et al. 1994]. Both patterns follow the idea of encapsulating requests into objects. Whenever a user calls a specific function of the application, the request is turned into a command object. The COMMAND PROCESSOR pattern illustrates more specifically how command objects are managed.
ACCEPTOR-CONNECTOR [Buschmann et al. 2007a]	Before peer event handlers in a networked system can execute their functionality with other peer event handlers they must first be connected and initialized. The connection establishment and initialization code of a peer event handler, however, is largely independent of the functionality that it performs.	Decouple the connection and initialization of peer event handlers in a networked system from the processing that these peers subsequently perform.
PUBLISHER-SUBSCRIBER [Buschmann et al. 1996]	A situation often arises in which data changes in one place, but many other components depend on this data. We are looking for a more general change-propagation mechanism that is applicable in many contexts. When some internal data element changes all clients that depend on this data have to be updated.	One dedicated component takes the role of the publisher. All components dependent on changes in the publisher are its subscribers. The publisher maintains a registry of currently subscribed components. Whenever the publisher changes state, it sends a notification to all its subscribers. Whenever a client wants to become a subscriber, it uses the subscribe interface offered by the publisher. Analogously, it can unsubscribe.
CLIENT PROXY [Buschmann et al. 2007a]	Accessing the services of a remote component requires the client side to use a specific data format and networking protocol. Hard-coding the format and protocol directly into the client application, however, makes it dependent on the remoteness of its collaboration partner, because invocations on remote components will differ from invocations on local components.	Provide a client proxy in the clients address space that is a surrogate for the remote component. The proxy provides the same interface as the remote component, and maps client invocations to the specific message format and protocol used to send these invocations across the network.
CACHING [Kircher and Jain 2004]	Repetitious acquisition, initialization, and release of the same resource causes unnecessary overhead. In situations in which the same component or multiple components of a system access the same resource, repetitious acquisition and initialization incurs cost in terms of CPU cycles and overall system performance. The cost of acquisition, access, and release of frequently used resources should be reduced to improve performance.	Temporarily store the resource in a fast-access buffer called a cache. To retain frequently-accessed resources and not release them helps to avoid the cost of re-acquisition and release of resources.
CONTAINER [Buschmann et al. 2007a]	Components implement self-contained business or infrastructure logic that can be used to compose applications. Since components may be deployed across a diverse range of applications and platforms, however, they cannot assume specific execution scenarios and technical environments.	Define a container to provide the execution environment for a component that supports the necessary technical infrastructure to integrate components into application-specific usage scenarios, and on specific system platforms, without tightly coupling the components with the applications or platforms.
EXPLICIT INTERFACE [Buschmann et al. 2007a]	A component represents a self-contained unit of functionality and deployment with a published usage protocol. Clients can use it as a building block in providing their own functionality. Direct access to the full component implementation, however, would make clients dependent on component internals, which ultimately increases application internal software coupling.	Separate the declared interface of a component from its implementation. Export the interface to the clients of the component, but keep its implementation private and location-transparent to the client.
WRAPPER [Gamma et al. 1994]	We want to add responsibilities to individual objects, not to an entire class. One way to add responsibilities is with inheritance. Inheriting attributes and methods from another class puts the inherited attributes and methods into every subclass instance. This is inflexible, however, because the choice of attributes and methods is made statically.	A more flexible approach is to enclose the component in another object that adds appropriate attributes and methods. The enclosing object is called a decorator. A decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients.

RESULT-CALLBACK [Voelter et al. 2004]	The client needs to be informed actively about results of asynchronously invoked operations on a remote object. That is, if the result becomes available to the REQUESTOR [Voelter et al. 2004] , the client wants to be informed immediately, so that it can react on the availability of the result. In the meantime the client executes concurrently.	Provide a callback-based interface for remote invocations on the client. Upon an invocation, the client passes a RESULT-CALLBACK object to the REQUESTOR . The invocation returns immediately after sending the invocation to the server. When the result is available, the distributed object middleware invokes a predefined operation on the RESULT-CALLBACK object, passing it the result of the invocation.
KEEP SESSION AT THE SERVER [Sor02]	Session specific data has to be stored in between requests, and made available to the code handling a request.	Keep all session specific data on the server. Keeping all data on the server and making sure it will newer leave the server, means you have no need to write elaborate error checking code to validate data every time it reenters the system from the client. It also frees you from implementing code that converts from the form the data is stored in while in the server (e.g. hierarchies of objects) to a form that can be transmitted over the wire between client and server.
SERIALIZER [Martin et al. 1997]	Every major application needs to read objects from and write them to a varying number of backends with different representation formats. Application classes should have no knowledge about the external representation format, which is used to represent their instances. Otherwise, introducing a new representation format or changing an old one would require changing almost every class in the whole system.	The Serializer pattern lets you efficiently store and retrieve objects from different backends, such as flat files, relational databases and RPC buffers.
PLUG-IN [Marquardt 1999]	An application that is required to be highly adaptable, or be extensible to support future functionality or modules. How can functionality be added late? How can the functionality be increased after shipping?	Factor out functionality, and place it in a separate component that is activated at run time. This component is called a PLUG-IN.
PLUG-IN REGISTRATION [Marquardt 1999]	Application has defined Framework Interfaces and Plug-In Definitions. Plug-Ins are available. User or application decides at run time which Plug-In to activate. How are the Plug-Ins known to the application?	The application defines a place where it looks for available Plug-Ins. Each Plug-In installs itself there.