

# Reactive Fabric Technology

White Paper Part 3/3 - Formalisation of The Technology

SDK For iOS, OSX (and coming for Linux)

Product White Paper

October 2019



"Language is not simply a reporting device for experience  
but a defining framework for it."

"A change in language can transform our appreciation of the cosmos"

*Benjamin Whorf, American Linguist 1897 - 1941*



Author: Terry Stillone ([terry@originware.com](mailto:terry@originware.com))

Web: [originware.com](http://originware.com)

Version: 1.0

## 🌀 Table Of Contents

Title	Description	Page
🌀 1. Introduction	Discussion on the content of this document and introduction to the core technology concepts.	3
🌀 2. Core Technology Concepts	Focus on the System Variance concept and the application of Element scoping.	4
🌀 3. The Design Methodology	Outlines the steps involved in synthesizing Designs. Direct examples provided to demonstrate the design methodology.	5 - 11
🌀 4. Aspects of Design	Focus on the various considerations that come into play during the Design process.	12
🌀 5. Test & Simulation Design	Focus on the design of Test and Simulation rigs.	13 - 15
🌀 6. Reactive Fabric Architecture	Focus on the SDK architecture of Elements, Notifications and Evaluation Queues.	16 - 17
🌀 7. Fabric Topologies	Outlines various Fabric topological types.	18
🌀 8. Mathematical Notation For Element Interactions.	Outlines the mathematical notation for Notification Spaces and the mapping between these spaces that represent Element interactions.	19
🌀 9. Scalability And Performance	Focus on performance analysis, bottleneck localisation and mitigation options.	20
🌀 10. Code Samples	Code examples of a collection of operators that demonstrate element handling of both data and control notifications.	21 - 24
🌀 11. Technology Side Aspects	Focus on hybrid Reactive Fabric systems and future possible uses of the technology.	25
🌀 Appendix 1. Reactive Fabric History and Origins	Focus on the pre-technologies and concepts that Reactive Fabric originates from.	26
🌀 More Information	Document links for the White Paper, Reactive Fabric Technology and the Originware site. Includes contact email links.	27

# 🌀 1. Introduction

## 1. Introduction to Part 3 of the White Paper.

In this section (**Part 3**), we go into the **technical formalisation** of **Reactive Fabric Technology**. If you have not read **Part 2** as yet, you need to at least read the **Reactive Fabric Element Patterns** (section 3) of **Part 2** before continuing here.

In summary, this content now goes on to the more technical aspects of system design by modelling examples of real world systems. We follow through the steps of the design methodology, discuss the architecture of the **Reactive Fabric SDK** and finish with some operator code examples.

While reading this document, please keep in mind that the material given here is quite dense and complex. The content covered in this White Paper is more worthy of fitting into a book and so there is a good deal of summarization. Examples are heavily used in this document to ground the given concepts and subjects. You may need to give your self a few separate reading sessions, not only to read this content, but to also integrate and appreciate the possibilities that the technology opens up.

## 2. Introduction to Core Technology Concepts.

**Reactive Fabric** operates on the "**Design for Interaction**" principle. It models interaction as the passing of **Notifications** between **Elements** (i.e. processing nodes) along assigned transport pathways. System behaviour (also termed "**System Process**") is therefore represented as the propagation of **Notifications** through the network (termed **Fabric**) of transport paths.

**System Process** has both **Structural** and **Temporal** aspects. Here,

**Structural** refers to            *"what is done"* and  
**Temporal** refers to            *"when it is done" or "the cycles it follows"* <sup>1</sup>.

The topology of the **Fabric** reflects the **Structural** nature of the **Design**. It reflects the inputs and the outputs of the system and what is performed in between. Separately, the **Scopes** in the **Design** are used to reflect the cycles and states of the system and so reflect **Temporal** aspects (note: **Scopes** are defined in the next section).

Transport pathways can support simple one-way messaging (between **Elements**) or more complex two-way interactions. Their directional nature contributes another layer of characteristic to the **Fabric** topology.

**Reactive Fabric** as a technology is multifaceted, not only is it a visual depiction of a Software System, it is also a development methodology. It incorporates design directly into the development process. It transforms design into an incremental process and synchronizes design cycles together with implementation cycles (which implies synchronization with test cycles as well).

The object of the technology is to decompose complex systems (including system-swarms) into simple interaction-designs and so it provides methods of analysis of system behaviour and decomposition. It also breaks down designs into manageable units by differentiating levels of behaviour (termed **LOB**):

- **Macro** level-behaviour designs depict high-level system management and orchestration.
- **Mid** level-behaviour designs reflect middle management, providing **Macro** designs with an interface to subsystems and bring an organisation to collective **Micro**-level behaviour. **Mid** designs also provide the pathways for funnelling and routing of the input and outputs of **Micro**-level behaviour.
- **Micro**-level behaviour designs define more of the specific fine detail processing behaviour.

---

<sup>1</sup> Cycles have both structural and temporal aspects.

## © 2. Core Technology Concepts

### 1. The Concept of System Variances.

Introducing the concept of the System Variance:

a **System Variance** is an intrinsic feature of a system that fundamentally varies the behaviour of the system and as such, it acts as a "**System Parameter**".

An important point to note here is that a **Variance** fundamentally changes system behaviour. So for example, a switch in a home entertainment system that changes the audio output between radio channels is not an implicit **Variance** but a switch that changes the output between the radio, music, TV, etc is, because it acts on the system to operate in a very differently way (and also operate with very different user controls). **Variances** express **temporal** system process discontinuities (which may also inflict **structural** changes to support those temporal qualities).

**Variances** may have constraints, they can be operative over a particular phase of the full system life cycle, such as at configuration time (these are termed: **Configuration Variances**), they can be operable over the operational cycle (**Operational Variances**) and they can be confined to particular subsystems rather than the whole system (termed: **Subsystem Variances**).

As a more specific example of a **Variance**, lets take look at the states of a **TCP/IP Network Interface**.

The interface has operational states of:

- ❄️ (i) Not Connected.
- ❄️ (ii) Connecting/Disconnecting (to/from IP address).
- ❄️ (iii) Connected to an IP address.

These connection states form the intrinsic "**State Space**" of the interface-subsystem and constitutes a **Subsystem Operational Variance**. The subsystem behaviour must differentiate between the given **Variance** states in order to perform appropriately, e.g. a send request to the **Network Interface** in the **connected-state** operates differently to a send request in the **not-connected-state**.

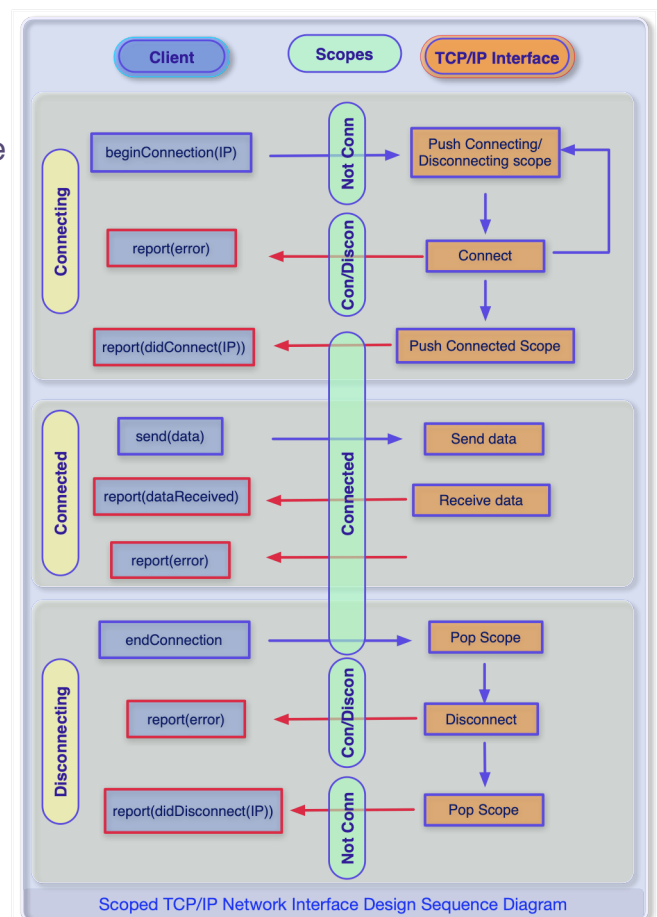
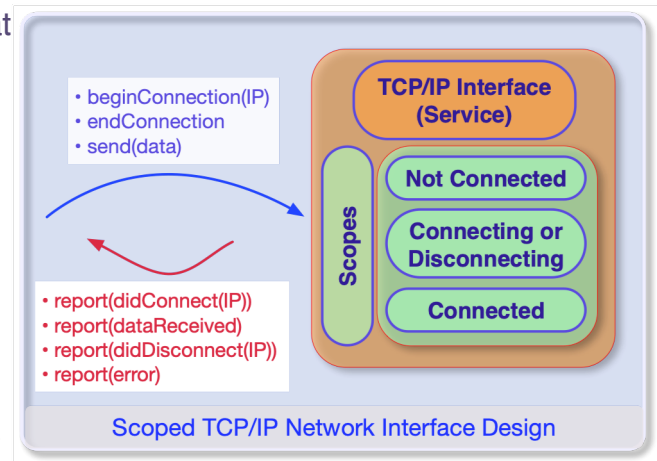
Rather than introduce distinct elements to handle the all the individual variance-states, **Reactive Fabric** applies the design concept of a "**Scope**". The **Scope** models the impact of the **Variance** on the associated **Element**. The element is modelled with the **Variance state-space** and it becomes part of its operational state.

Following through with the **Network Interface** example, this element has an operational state transition set of:

$$(i) \Leftrightarrow (ii) \Leftrightarrow (iii)$$

which functionally equates to a stack **push** and **pop** pattern. So the **Scoping** mechanism within the element is implemented (in code) as a stack of **Element-handlers** and on initiation the **Element** operates with an **Element-handler** corresponding to state (i). On reception of a connection request, it pushes the **Element-handler** for state (ii), which engages connection negotiation. On connection establishment the (iii) **Element-handler** is pushed to provide connection behaviour. The reverse action is performed during the disconnection scenario.

All this element-behaviour is modelled in the design and sequence diagram to the right:



## 📍 3. The Design Methodology

### 1. Design Objectives.

Let's clarify our design process objectives, the goal is to produce formal, implementable designs. Towards that goal, a number of intermediate designs are authored. The first set is **Conceptual** in nature which are then migrated to **Practical** (which are applied for use). **Conceptual** designs are used as a guide to decompose the system into element quanta and from that, determine team(s) to work on specific design sections. Group meetings generally work on **Macro** designs while individual teams work on specific **Mid** and **Micro** designs. Group meetings are also responsible for the back-integration of team **Mid** to **Micro** designs together with younger group **Macro** level designs.

A design of course starts from a blank slate. It first precipitates as a simple, raw **Macro, Operational** design, probably drawn on paper, which then undergoes iterative evolution. It is refined until some level of group consensus is reached as to its adequacy. This design becomes the first tagged "**Conceptual Operational Design**".

#### Note

Formal designs are tagged with their type and iteration, for example the first formal design is normally: **Conceptual Macro-Operational Design Iteration 1** which normally leads onto the **Practical Macro-Operational Design Iteration 1** version.

### 2. The Various Design Sets.

The first project **Design** will probably focus only on the **Operational** design aspect. This is really just one facet of the whole system design set. The more complete design set includes:

- 🔗 The **System Life Cycle** design: addresses the cycles of construction and destruction.
- 🔗 The **System Operational** design: addresses the business logic and target behaviour.
- 🔗 The **Error/Failure/Failover** design: addresses robustness and handling failure cases.
- 🔗 The **Performance** design.

### 3. Initiating Operational Design.

While the design space is generally overflowing with virtual possibilities, there are some fixed points that need to be identified before beginning design. These include:

- Synthesizing system inputs, outputs and IO ports.
- Synthesizing the processing required for post-processing inputs and pre-processing outputs.
- The processing requirements for IO ports, such as API handling, security, encryption etc.
- Synthesizing OS Service dependencies, such as Presentation, Location, Audio, Video, etc and the processing requirements for employing these services.
- Identification of abstractions such as View Models, Data Store Models, etc.

In the first step of authoring of the **Conceptual Design**. We ask the questions:

- **Who or what initiates system interaction?**
- **What** are the **Use Cases** that are engaged by that initiator?

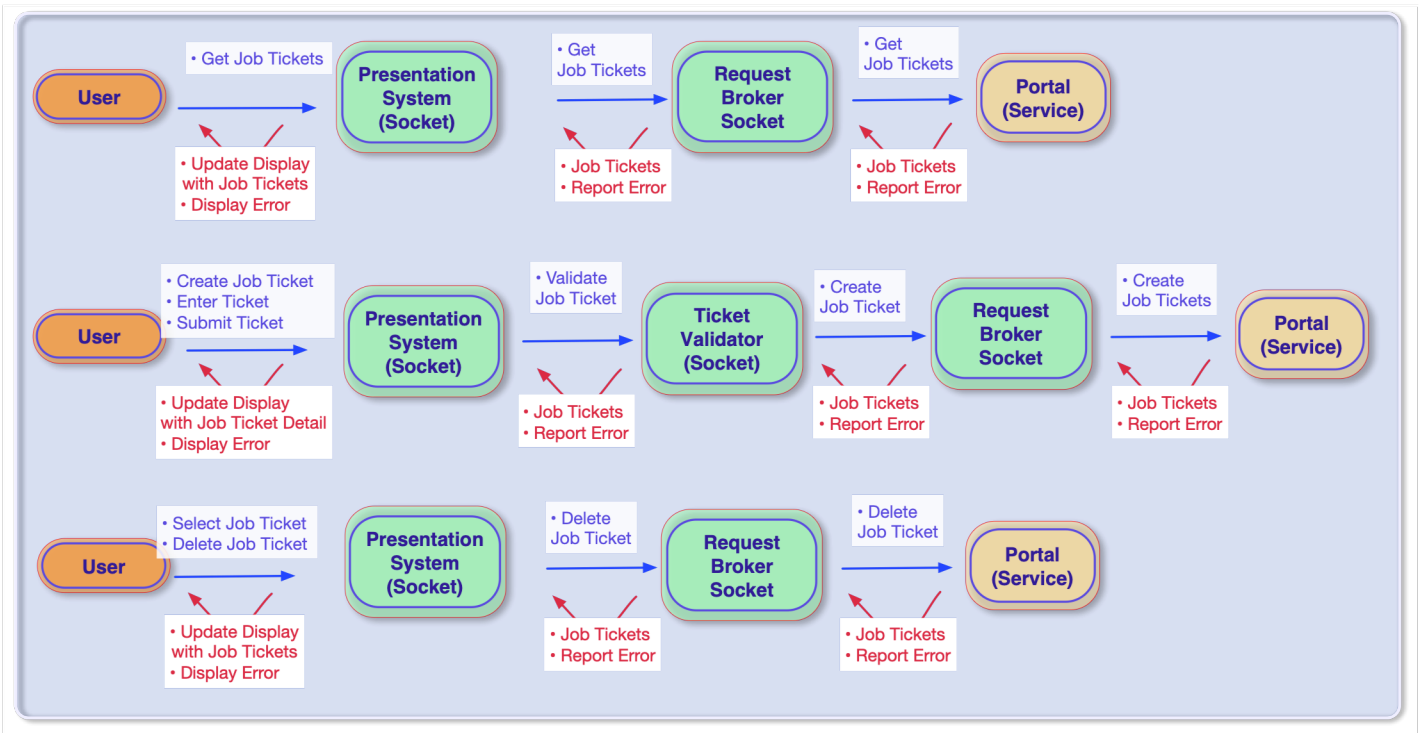
Let's use an example, a hypothetical mobile **App** that manages **Job Tickets** through some web portal. The **App** communicates with the net portal for **Job Ticket** information and is to:

- **Display** the users submitted Job Tickets. This is the default display.
- **Create** new Job Tickets, in a popover display.
- **Rescind** existing user Job Tickets, by selecting jobs in the default display.

### 3. The Design Methodology (Cont.)

#### Step 1: Author a first Draft Macro Conceptual Design.

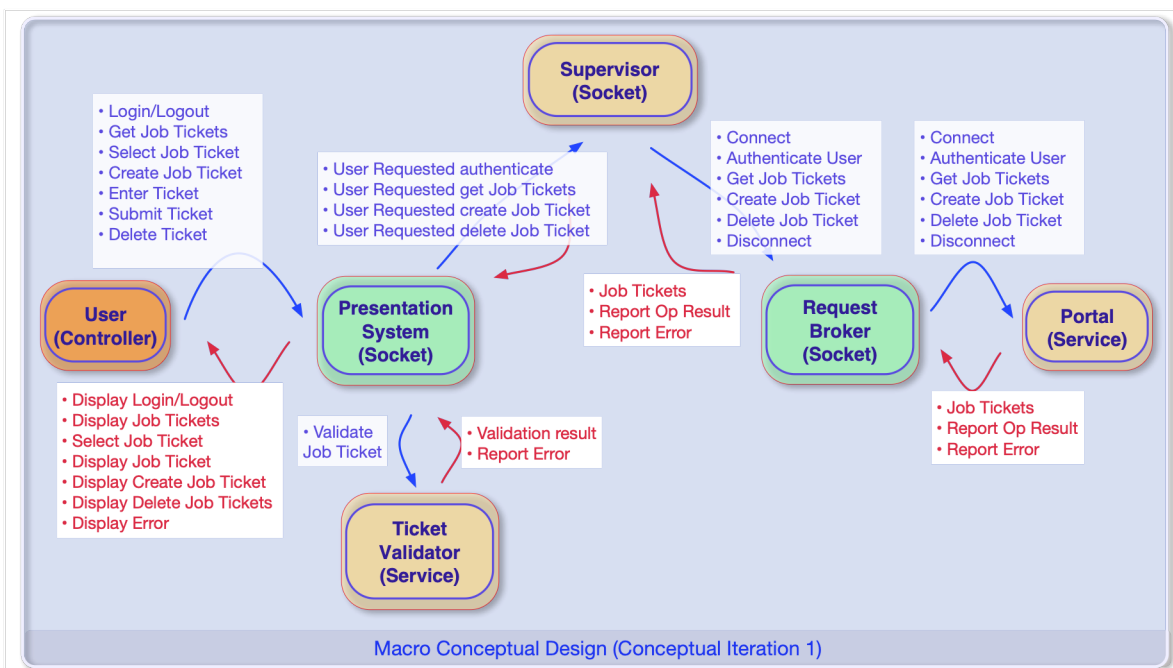
Here is a first draft of the capture of the Apps core business use cases with the **User** as the initiator:



**Note**

We did not capture the user authentication use-case in this initial draft, it should have been recognized but the group got caught up in side issues during the meeting. In this scenario, unresolved aspects would then be addressed in followup design meetings and later incorporated into design. The point here is that the design process is self supporting and provides flexibility to cover situations that arise during development.

From this point in our design process, we add concerns for the operational supervision of the whole system and we consolidate the separate use cases given above into a common design to arrive at the first draft **Operational Conceptual** design, given here:



**Note**

Authored designs are normally supported with an additional interaction diagram to add more process context, for brevity it is not given here in this example.

## @ 3.The Design Methodology (Cont.)

### Step 1: Author a first Draft Macro Conceptual Design (Cont).

Lets assess the elements in our newly authored design for complexity and internal structure:

- The **Supervisor** mediates between the Presentation subsystem and the Request Broker. Much of its work is merely passing over notifications, so it is quite simple.
- The **Request Broker** and **Ticket Validator** are simple enough and will probably end up being single elements in themselves.
- The **Portal Service** is complex and will require sub-elements to handle **API**, **HTTP** session protocol and possibly **TCP/IP** connectivity depending on OS support. So this subsystem will most likely comprise of two to three elements.
- **The Presentation** element is a pivotal subsystem in the whole design. In terms of its element count it comes down to good or bad design.

Returning to the design process, teams are now assigned to work on the individual **Mid**-level designs. Here is the assignment:

(Team 1) The **Presentation System & Ticket Validator** subsystem.

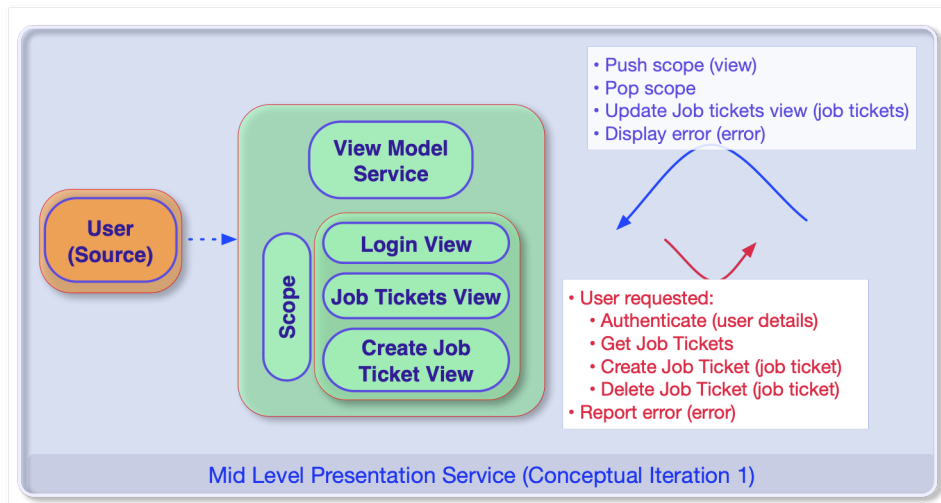
(Team 2) The **Portal Service** subsystem.

(Team 3) The **Request Broker & Supervisor** subsystem.

### Step 2: Produce Mid-level Conceptual Designs.

Each team now goes on to develop the **Mid**-level designs for their respective subsystems. We will concentrate (only) on the **Presentation** sub-system in this document but I will mention that the **Portal Subsystem** ends up being a pipeline of an **API (Socket** element) going to a **HTTP Session (Service** element).

The **Presentation** system once assessed, (in **Mid**-design level terms) is replaced by a more detailed **View Model Service** that includes scoping for each of the views to be presented. Here it is:



This new design also implies alterations to the original **Macro** design. First, the **Ticket Validator** element is to go and validation will now be integrated into the **Create Job Ticket View** scope logic. Second, the **Presentation** element was previously a **Socket** and now the replacement **View Model** is now a **Service**, driven by the **Supervisor**, which implies flow-on changes to the **Supervisor** character (i.e. it now becomes a **Controller** element). Finally, **User** interaction becomes implied, that is not direct as the **OS Presentation** system now handles user input internally within the **View Model Service** element itself. This is depicted in the design with a dotted pathway.

### 3. The Design Methodology (Cont.)

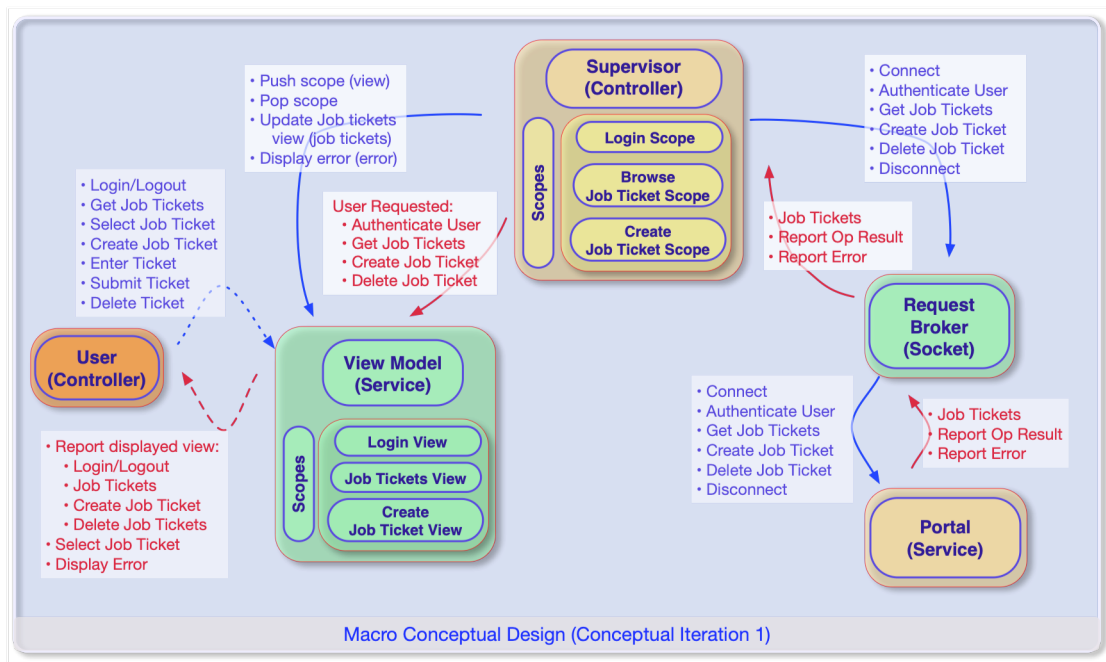
#### Step 3: Integrate Mid and Macro Designs to Form Conceptual Macro Design (Iteration 1).

The next design-process objective is the back-integration of the new **Mid** level designs with the original **Macro** design to form the first formal **Macro** conceptual design (to be tagged as **Conceptual Macro-Operational-Design Iteration 1**). The design meeting group convenes to undertake this and applies the knowledge gained and impacts formulated from the work on the **Mid** level designs. As a consequence of the **View Model Service** being scoped, it is decided to sympathize the **Supervisor** with the **Operational** scopes of:

- ❄ The Login Scope (which will direct the Login view to be presented).
- ❄ The Connected Scope (which will direct the Job Tickets view to be presented).
- ❄ The Create Job Ticket Scope (which will direct the Job Ticket Entry view to be presented).

The **Supervisor** element is also changed to a **Controller** pattern so that it directly manages the **View Model** and the **Portal Services**.

The final **Macro Conceptual** design version is given here:



#### Note

While altering element pattern types (such as with the **Supervisor** here) may appear subtle at this phase, it alters causal initiation and that has deeper implications later on.

#### Step 4: The Practical Macro-Operational Design (Iteration 1).

Drawing from the **Conceptual** design, the design team moves to derive the first **Practical** design. Where **Conceptual** design prioritises on the **pure functional aspects** of the system, **Practical** design focuses on incremental design and the design-implementation-test cycle. So the **Practical** design, draws from the **Conceptual** design but only incorporates what is to be implemented in the current iteration.

For our scenario, **Practical Iteration 1** only facilitates capability for login and logout in its design (i.e. scopes related to login and logout). Correspondingly, the **Portal Service** will only supply enough capability to connect, authenticate users, disconnect and log-out.

The remaining capabilities such as the **Ticket Display** (Connected/Job Tickets View scopes) will be introduced in **Practical Iteration 2** and the ticket **Creation** and **Deletion** in **Practical Iteration 3**.



## 3. The Design Methodology (Cont.)

### 3. Additional Design Cycles.

The designs we have produced up to this point have only addressed **Operational** design. At some milestone, **Failure** design and system **Lifecycle** design must be also addressed. When they are actually considered, depends on project priorities. If the system has a high security/reliability priority then they will probably discussed early in the project cycle. For situations that have a user feedback priority and need to get a system up and running as soon as possible, it may be delayed to after the basic system is implemented.

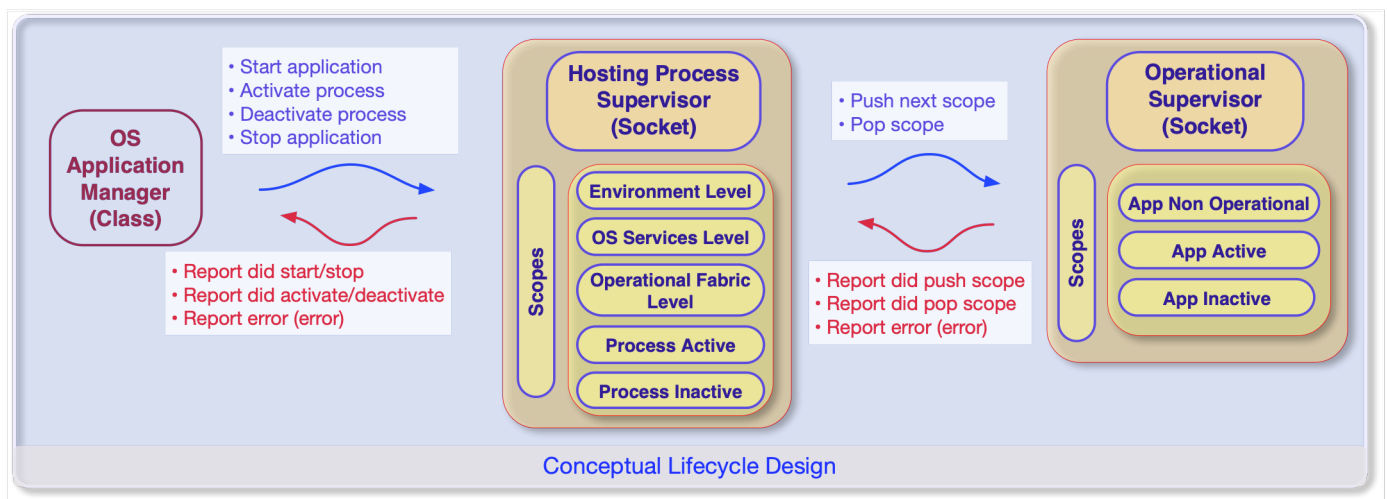
Systems that require intensive computation and verge on performance concerns may need to institute a project **Performance Design** iteration cycle.

### 4. Lifecycle Design.

**Lifecycle Design** in general, addresses the environment that the **Operational Design** operates within. It focuses on concerns related to:

- Boot-up security and environmental integrity checking.
- Hosting process modes (e.g. foregrounding and backgrounding modes, containers etc).
- Assessing availability of OS services to fully support the system.
- Initiation and finalisation of OS Services such as the Presentation system, Camera system etc.
- Construction and destruction of the **Fabric**, i.e. the **Operational Design** elements.
- Deployment of the **Fabric**.

Below is the **Lifecycle Design** for our **Job Ticket App**. The design handles **Fabric** construction / destruction, bringing the **Operational Design** up / down and managing **OS Process** states. In the design, the **OS Application Manager** (in class form or code) drives the whole **System** and its associated process states (i.e. **App** foregrounding (activation) and backgrounding (deactivation)).



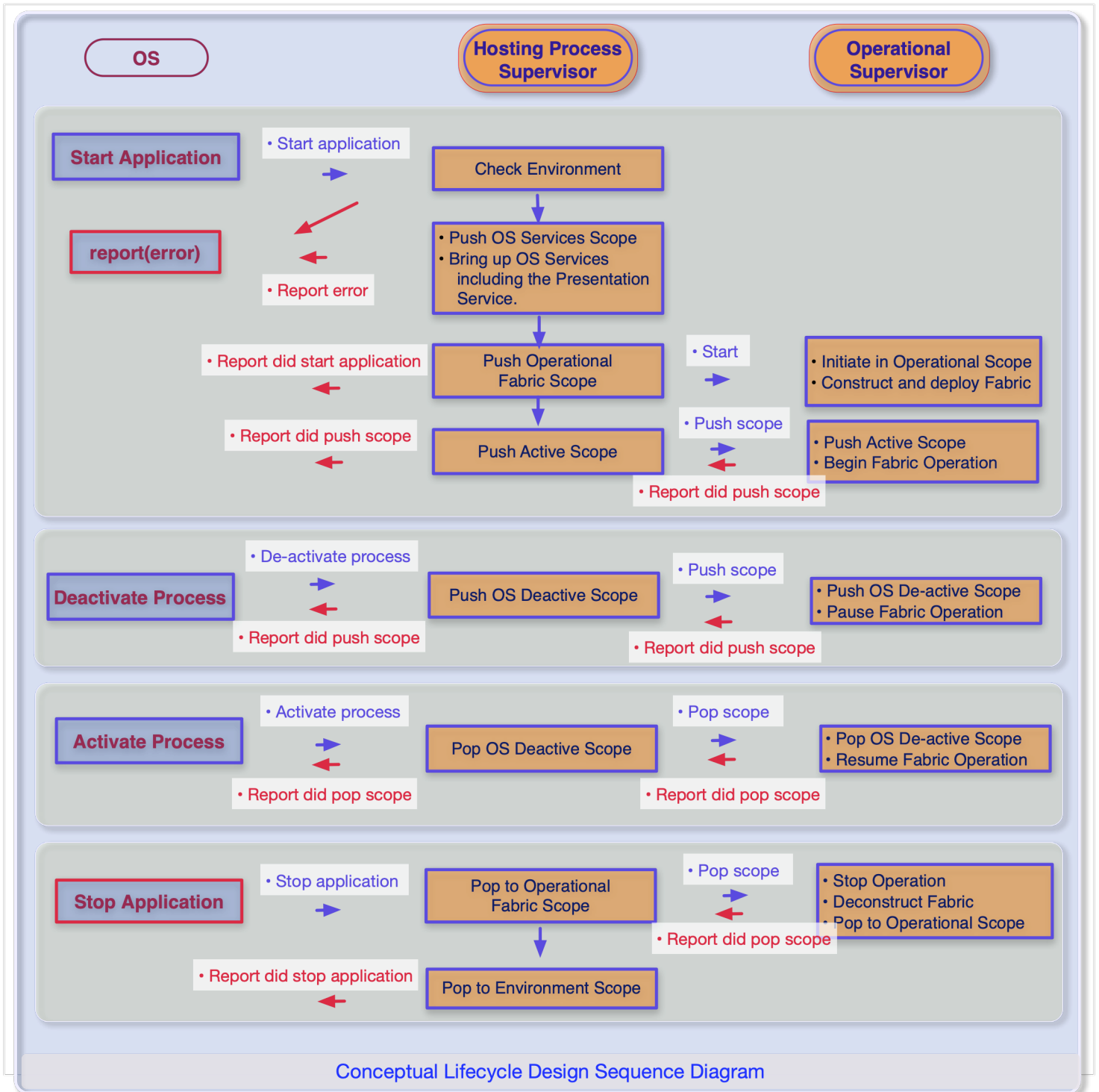
From this design, the internal process for bringing the **App** up is now:

- The **OS Application Manager** emits a **Start Application** notification to the **Hosting Process Supervisor Socket**.
- The **Hosting Process Supervisor** then auto pushes its various scopes in sequence, performing the appropriate actions for their respective scopes until it gets to the **Process Active** level and then emits a push **App Active** scope notification to the **Operational Supervisor** element.
- The **Operational Supervisor** in turn, initiates our previously given **Operational Design**, which engages the actual **Job Ticket App** behaviour.

## 3. The Design Methodology (Cont.)

### 4. Lifecycle Design (Cont.)

Here is a sequence diagram to detail the interactions for the **Lifecycle** design:



Summarizing this **Lifecycle Design**, there are a number of temporal cycles incorporated into element scopes, these being:

- (i) The **Start** and **Stop** cycle.
- (ii) The **Operational Fabric** construction and destruction cycle.
- (iii) The App **Foreground** and **Background** cycle.

The **Hosting Process Supervisor** observes all these cycles and the **Operational Supervisor** in sympathetic synchronization, observes cycle sets (ii) and (iii).

## © 3.The Design Methodology (Cont.)

### 6. Appreciation Of Pure Design.

Lets reflect back on this whole design set and note some of the key points that came out of the design process:

- The **Lifecycle Design** exhibits **scope symmetry** in its progression of from-initiation and to-finalisation. As a general rule, if **structure** exhibits **symmetry** then it is assumed that in turn, **process** will also reflect that **symmetry**.
- **Synchronisation of Scopes** between elements is another key design principle as it imbues to the system with a symphony of complex macro behaviour through the engagement of simple actions.
- **Balance** is an important design aspect, each scope in the lifecycle design is fairly simple (in their responsibility/behaviour) and that behaviour is reasonably **balanced** between elements and scopes.
- **Handling system complexity** at the **design level** is more fluid to work with, rather than at the code level.
- Generally, locating where behaviour would be naturally expressed is probably the ultimate determiner of a good design. This goes in hand with acknowledging the **scope** and **dimensions of behaviour**, determining its structural scope (local, subsystem, system) and its temporal scope (specific lifecycle, operational and full system lifecycle).
- Through the acknowledgment of the dependants of the behaviour (both upstream and downstream), the natural character of the behaviour becomes more evident. Relevant questions in the process include "**what controls**" this behaviour and "**what uses**" the products of this behaviour. These design principles project onto cleaner code with distinct responsibilities and boundaries.
- **Scopes** will tend to reflect the natural cycles of the system and these become obvious process contour-boundaries for segregation of practical design versions and implementations. This is an example of the inherent power of **Reactive Fabric**, to draw out natural process contours (i.e. discontinuities) which then appropriately flow onto the design and impact the development process so as to have the development process also reflect those natural contours.
- **Practical** designs are derived from their **Conceptual** counterparts. That does not mean that when it comes time to author a particular **Practical** design it should strictly follow the **Conceptual** ancestor. First there may have been insights gained between the two authorings that require alterations to the **Conceptual** version and second the **Practical** may need to deviate from the **Conceptual**. It may possibly co-align in later versions and it may not.

At this point you may be arguing that this scheme is really a pseudo type of "incremental design" and you are probably correct. Producing a full **Conceptual** design ahead of **Practical** designs is not pure incremental design. What **Conceptual** designs are there for, is to gain **forward knowledge** rather than at the cost of learning it later and necessitating major re-design. **Conceptual** designs also size the project and indicate resource requirements. But, the designer should also recognize their limitations, while **Conceptual** designs provide a good indication of **Structural** design issues they may not give an adequate description of **Temporal** issues. Keep in mind, there is no single design solution. There are a set of solutions with some fitting better for the project, product requirements and team expertise. The **Design Space** is a canvas upon which designers actively explore evolving possibilities.

**Reactive Fabric** works by operating on simple and natural principles. This approach can appear to be naive or even whimsical with respect to real world systems. If you are developing on the principle of "**designing for code**" (and not "**pure design**") you are probably not realising, pure simple ways of doing things and introducing artificial complexities and boundaries. These aspects of simplicity and naturalness (of expression) also drives the inherent power obtained in the use of **Reactive Fabric**. This may not be immediately apparent and require direct experience with incremental design cycles to fully appreciate the value.

## © 4. The Aspects of Design

### 1. The Dimensions of Design.

Design is a complex and confusing process. There are many aspects and considerations to collect, assess, draw relationships with and integrate when designing. Each project is also unique in its priorities, constraints and complexities. The core of these dimensional aspects include:

- System behavioural complexity and how to decompose that complexity.
- Dependencies (attached/peer devices, third party libraries, OS dependencies).
- Requirements: Plug-ability, Configurability, Adaptability and Robustness (under changing conditions).
- Product quality assurance: Testability and Simulation-ability.
- Product future: Maintainability and Extendability for future evolution.
- Constraints (e.g. performance, memory foot print (for restrictive IOT systems), developmental personnel and resources).

Designs need to balance the various project aspects in order to provide the best design fit. That relies on on-going conversation with context to priorities and trade-offs. **Reactive Fabric** provides some support with its ability to breakdown designs through:

- Incremental design cycles (and handle high priority concerns first).
- Behaviour levels: Macro/Mid/Micro (and decide which levels are actually relevant for the project and which of those levels are to address what concerns).

There also some general design guidelines that may assist:

- **Identify natural structural and temporal boundaries:** an authentic design will reflect the structural and temporal boundaries of the real system. If two system facets are independent, then in the design they will not be connected and probably topologically distant. If two facets are related or dependant, that type of relationship (structural and temporal) should be reflected in their design notification pathways.
- **Identify System Variances.**
- **Respect symmetries and asymmetries:** scope structural symmetries should be reflected in their process (in how they behave and how they are driven).
- **Balance designs:** so that the various behaviour levels equally share the complexity.
- **Only give elements state if absolutely required:** functional behaviour is preferential. Element operation equates to a function mapping from their reception Notification space to their emission Notification space. There are cases where the mapping is dependant on the element state/history, for example an Element that is aggregating its received notifications it must store the aggregation result over its lifecycle. **Sockets** with bi-way interactions will either perform pass-through of replies (and be stateless with respect to replies) or will alter their behaviour in response to replies (and so be stateful). Also keep in mind, **Element** state does incur additional testing overhead.
- **Scoping inherently imparts state to elements.**
- **Use pathways for their direct intended purpose:** There can be design pressure to use a notification pathway for what it is not really mean't for, because it appears to be expedient at the time. Over role-use of a pathway confuses **actual** dependencies from their **idealisation**. That does not mean it should not be undertaken, but it should be assessed for alternate options.
- **Assess pathways with high load:** Pathways generally support both design **control** notifications and design **data** notifications. **Control** notifications should be designed to have a low frequency impact that is, to be low use high-level commands. If **data** notifications have an expectation of high load/frequency then, there are many options for design mitigation, please see: *Section 9: Scalability and Performance*.

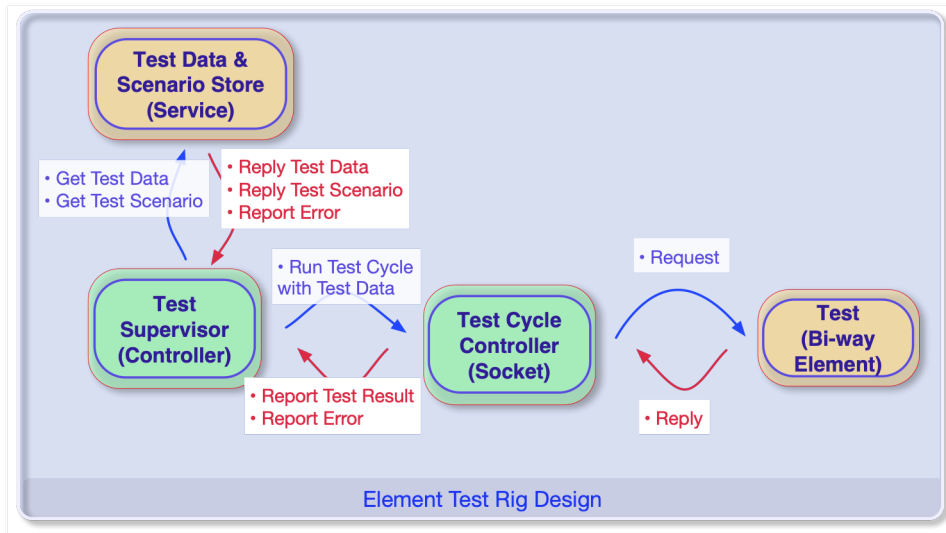
## @ 5. Testing and Simulation Design

### 1. Designing Test Rigs.

**Reactive Fabric** is a natural environment to perform unit testing, system testing and simulation as these systems in themselves constitute interactions characterised by:

Controller  $\Leftrightarrow$  Test Element

More support is of course required as the controller does additionally need test data for the test runs and supervision to handle test cycle result aggregation. So a test rig is required and the general design for the rig looks like:



In this design, the **Test Supervisor** manages the whole testing process while the **Test Cycle Controller** is only responsible for a single test cycle. The **Test Supervisor** fetches the relevant test data, expected test results and test scenarios and then feeds these to the **Test Cycle Controller**. The **Test Cycle Controller** in turn feeds test notifications to the target **Element**, compares replies with expected results and replies back with the cycle result. (If the test **Element** is a single-way notification type, then an Element wrapper is required to feed input notifications and collect emitted notifications).

The first choice for test-data is to generate it and also generate the expected results. Test-data can also be captured from the actual system. This is handy when Element behaviour is time dependant and the test results are time sensitive. This leads onto simulation scenarios and discussed in the next section.

The test target can take a number of forms:

- a single **Element** or **Subsystem Element**.
- an **Element Expression** .
- a **Whole System** with minimal separate inputs and outputs.
- a **System Swarm**.
- a modelled **Network**.

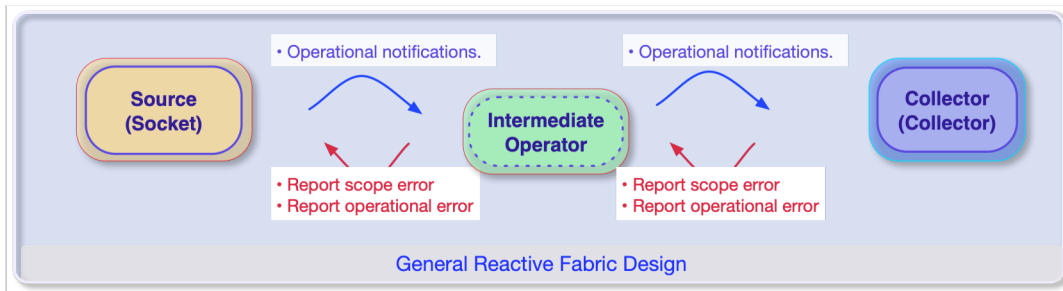
#### Note

Testing is an offline approach to checking **Element** conformance. There are also inline (i.e. in-**Element**) approaches can also be employed. These perform checks within the **Element** itself for: reception and emission notification conformance, notification bandwidth conformance, etc. These **Element** checks can be conditionally compiled in, or virtually hosted by the **Element**.

## @ 5. Testing and Simulation Design (Cont.)

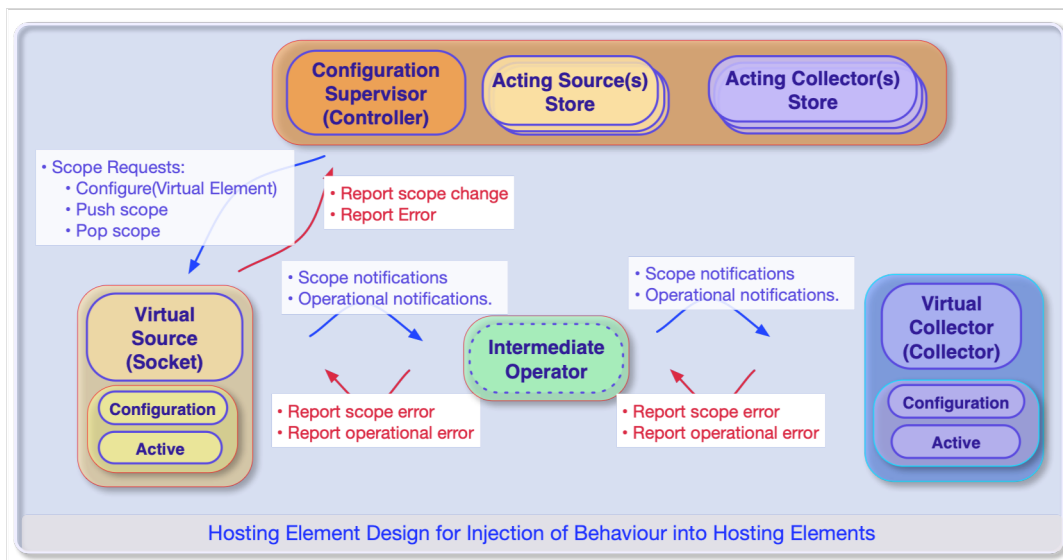
### 2. Collecting Simulation Data From a System.

The first step in simulating or testing with recorded data is to setup the target system to record both input and output notifications. Lets take a generic example of a **Reactive Fabric** expression:



The **Inputs** and **Outputs Elements** that comprise the **boundary** of the **Fabric** become the prime targets for recording realtime **Notifications**. They also will be subsequently used to replay the recorded notifications. The capture-inputs must be made configurable, so to be switched between real input and playing pre-recorded input.

There are many possible switching designs but for here we will center on a **Virtual Element Design** as it again **preserves the native topology** of the target system design. To allow recording and monitoring, the **Source** and **Collector** must be made configurable, taking on a **Configuration** and **Active** scope state-space. A **Configuration Controller** element is added, which houses the all the **Acting Source Elements(s)** and **Acting Collector Element(s)** for the complete set of target scenarios. The capture-enabled design now becomes:



In this design: the **Configuration Controller** emits the designated active **Elements** in a **Configure Notification** while the system is in the **Configuration** scope. Those target **Elements** then capture their respective **Acting Elements**, to engage and then employ them during the **Active** scope cycle.

So, during operation the **Fabric** is first configured for the target environment, for a **production scenario**, it is configured with production active **Elements** and for a **capture scenario** it is configured with capturing active **Elements**.

The **Acting Source Element** set comprises of:

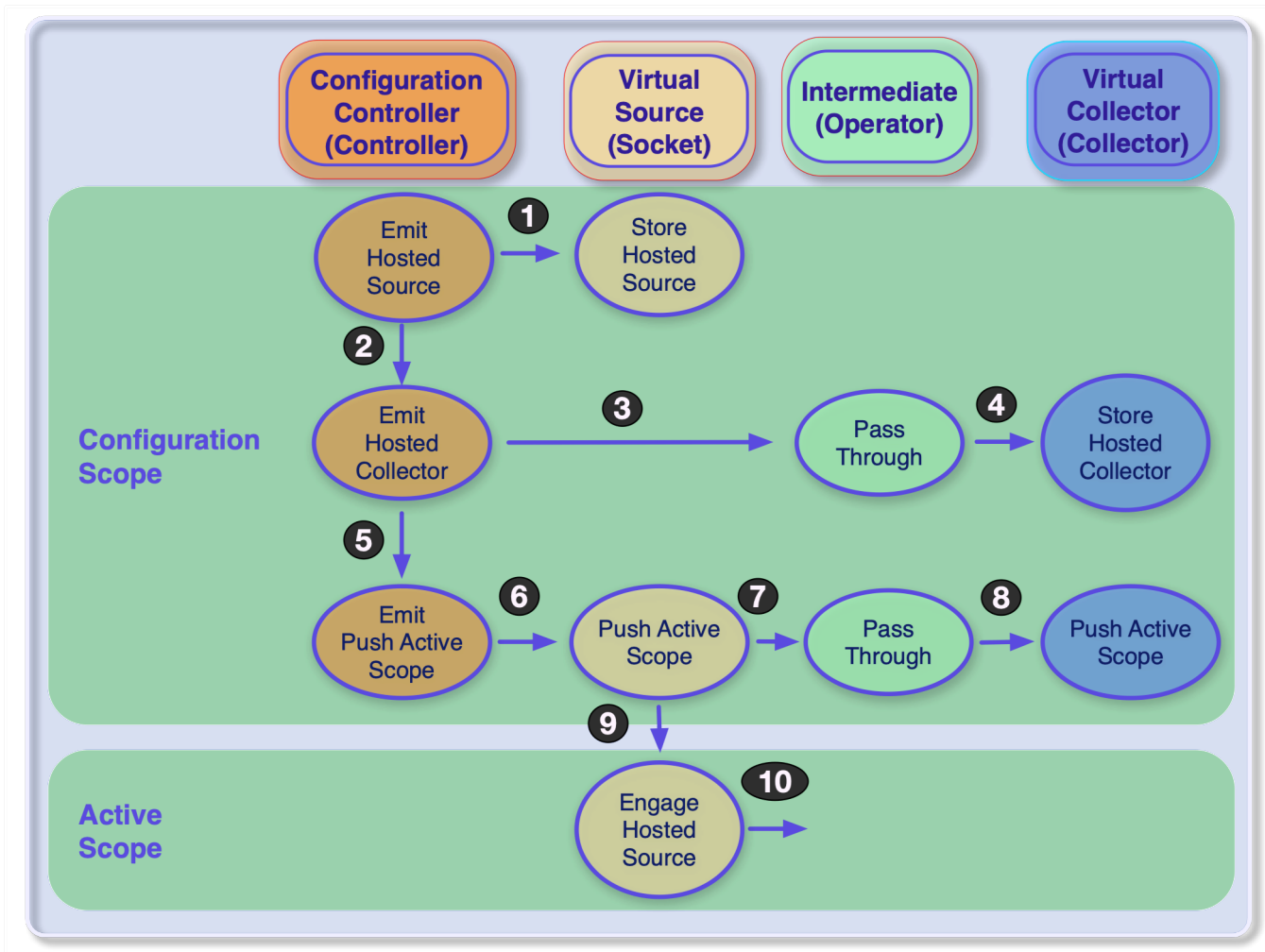
- The native **Source Element**, which processes real input.
- The recording **Source**, which could be a unique element, the **Source** with internal switching to record or a combination of the native **Source** composed with a recorder **Operator**.
- The **Player Source Element**, which takes input from the recorded notifications.

The **Collector** elements (which are to record resultant notifications) are handled in a similar fashion. They become **Virtual Elements** and are configured with their acting collector behaviour depending on the scenario.

## 5. Test and Simulation Design (Cont.)

### 2. Collecting Simulation Data From a System (Cont.)

Here is the sequence diagram for our capture-enabled virtual element design:



Recording and archiving input notifications can have enormous benefits for testing but that data may have a limited lifetime and become invalid over implementation cycles that alter basic system behaviour. So it is important to be able to automate the recording procedure so that recorded products can be easily refreshed.

Test Data notifications can also be procedurally generated, passed to **Test Cycle Controllers** and then used to stress test implementations. This is the initial choice of **Testing** design. **Test Cycle Controllers** can also be directed to randomly perturb notification timing so as to weed out race conditions. **Reactive Fabric's** own synchronization model (discussed on the next page) generally ensures synchronisation integrity but there are cases where elements in an expression are running in multiple and varied threads. A good example of this is a **join** or **zip** element sourcing from a number of independent sources. **Reactive Fabric** has an element thread hand over mechanism for these cases, that if observed will not have issues but if omitted, can appear to work while being a vulnerable future failure point. If reliability is critical, then stress and race condition testing should be observed.

#### Note

There may also be additional business case scenarios that can benefit from a **Virtual Element** design. For example, imagine a **purchase-charging** business case that may need to download charging models from a server to match the users regional tax regulations. The **Fabric** can download and configure the **Virtual** charging expressions during the configuration scope-phase and then apply those changing models in the **Active** scope. In these **Virtual Element** designs, the **topology** reflects the general expression nature and the individual elements express the specific required calculation (i.e. handle the more **Micro-level**

## 🌀 6. The Reactive Fabric Architecture.

### 1. Concurrency Model: Element Synchronisation and Evaluation Ordering.

**Reactive Fabric** assigns a dedicated **Evaluation Queue** to each evaluation expression chain. An assigned thread (attached to the **Evaluation Queue**) executes all the behaviour of the expression, including internal element processing and notification propagation along the element chain. This arrangement ensures the synchronisation integrity of expression evaluation.

The **Evaluation Queue's** role is to execute submitted jobs. Internally, it is composed of a **Job Queue**, to order and buffer submitted jobs and a **Thread** (or thread pool) to execute those jobs. So **Jobs** submitted to the **Evaluation Queue** are pushed onto the **Job Queue** and then are popped off and executed in the associated **Thread** in their queue order. This guarantees that individual calculations for the expression are executed in the order that they are submitted. When other expressions notify the current expression, those notification-calls are also passed to the **Evaluation Queue** so as to be performed in executional order.

The **Evaluation Queue** is also enhanced with special anti-deadlocking support to handle self-recursive calls on the same **Evaluation Queue**.

Inner **Element** co-operations such as timer actions are normally run in their own timer **Evaluation Queue** but their results are handed over to (termed "joined with") the element **Evaluation Queue** to again guarantee the timer products are synchronized with the element evaluation. Additional **Evaluation Queues** can also be employed within elements for multi-thread processing. Their products must also be joined with the element **Evaluation Queue** for notification synchronization.

### 2. Expression Evaluation Types.

Expression evaluation can be performed in a number of executional modes to suit concurrency needs. Some expressions may need to be asynchronous with respect to their caller, while others may require synchronous relationships (to their caller).

The full expression evaluation types are:

- 🌀 **Asynchronous Evaluation:** the expression is executed **asynchronously** in a separate **Evaluation Queue** with the caller not being blocked.
- 🌀 **Synchronous Evaluation:** the expression is executed **asynchronously** in a separate **Evaluation Queue** while blocking the caller until evaluation completion.
- 🌀 **Inline Evaluation:** the expression is executed **synchronously** in the same **Evaluation Queue** as the caller. Expression evaluation returns back to the caller when finished.

### 3. The SDK Notification Mechanism.

Notifications are performed in the **SDK** by calling a **notify()** method on the target **Element** class and passing the emission **Notification** as a parameter to the method. Each element in an expression chain calls the **notify()** method of their downstream element. This in effect maps the **notification chain** in **Design Space** to a **function call chain** in **Code Space**. The simplicity of this notification mechanism yields performance benefits while providing ease of debugging (by placing breakpoints at strategic **notify()** points and observing the call stack).

### 3. Coordinating Fabric Activity.

Normally, individual **Fabric** expressions are performed in a semi-independently fashion with expressions coupling to their co-dependant partner expressions. But there are scenarios where independent expressions may need to be co-orchestrated for purposes of collective throttling, such as mating with energy consumption profiles, adaption to low resources, etc. One design option is the use of a **Common Clock** to drive expression collectives. Here a **Source** element driven by a timer forms a **Common Clock** which emits a **Clock Notification** that feeds into the fabric **Sources** (which now are promoted to **Operators** in the new design) that triggers them to perform their own respective emissions. **Clock Notification** emission is then managed to match the required timing profile.



## 📍 6. The Reactive Fabric Architecture (Cont).

### 4. Virtual Time References and Scheduling.

Modern systems employ timer coalescing to perturb system timer events so as to co-align near-adjacent events and so reduce system energy demands. This severely impacts on timer ordering in a complex system as no longer does a timer triggered event strictly correspond to the system clock and various sub-systems can no longer rely on consistent relative timing of events.

**Reactive Fabric** makes use of **Virtual Time Schedulers** to ensure time values are in order and are comparable across disparate **Elements** and **Fabrics**. Notifications are timestamped with a **Virtual Time** reference rather than the real time. This timestamp reference does loosely correspond to the real time but is perturbed to guarantee order consistency across the **Fabric**.

**Reactive Fabric** is not temporally-strict enough for proper **Realtime** control but it is certainly suitable for device control where there is some temporal leeway, where the order of magnitude of the leeway roughly equates to the OS system timer coalescing leeway.

### 5. Notification Architecture.

**Notifications** in **Reactive Fabric** are an encapsulation of both internal (**SDK**) **Control** and **Design** notification payload tokens. These internal **Control** tokens signify pathway lifecycle events such as:

- The **Initiation** and **Completion** of the notification pathway stream.
- **Error** conditions raised by an element terminating the stream.

The actual design **Notifications** (that the designer works with) are emitted between the initiation and completion **Control** tokens. The **SDK** performs checks that elements conform to these rules.

**Notifications** may also be timestamped with their emission virtual time and are made available to **Element** processing for use in temporal decision logic.

### 6. Notification Pathway Architecture.

**Notification Pathways** can be either **one-way** or **bi-way** transport mechanisms. The forward emit path is always a **push mechanism**, pushing from the **emitting Element** to the **receiving Element**. In the **SDK** this is effected by a method call on the target element class while passing the **Notification** as a parameter.

The **SDK** has no direct synchronous **pull mechanism** but instead, there is support for an indirect asynchronous method of pulling notifications upstream called the **reply-back mechanism**. **Bi-way** transport pathways use this facility to pass reply-notifications upstream. The **reply-back-mechanism** requires the forward notification to additionally emit the **reply-back-mechanism** (which is implemented as a closure) together with the forward notification data and then in turn, the target element exercises the **reply-back-mechanism** to hand back replies.

### 7. Memory Sizes of SDK Elements.

Elements in the **Reactive Fabric** incur minimal memory footprint. As a loose guide:

- 📍 A non-scoped, non-subsystem Element maps to: **1 x class**.
- 📍 A non-scoped, Subsystem Element maps to: **3 x classes**.
- 📍 A a scoped Element maps to: **(Element class count) + (1 x stack) + (scope-count x struct)**.
- 📍 A Notification maps to An **enum** with an associated type which includes the **notification data item**.

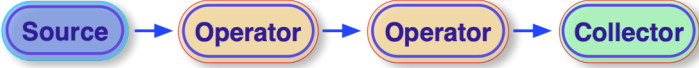
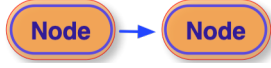
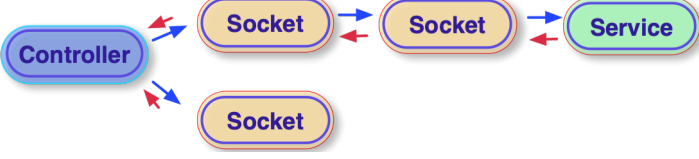
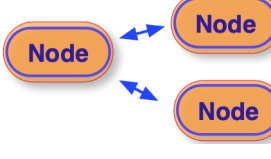
## 7. Fabric Topologies

### 1. Classification of Topological Types.

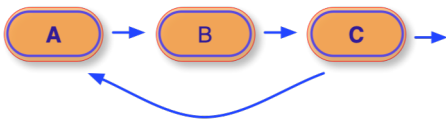

In order to visualise **pure topological nature** we need another more basic visual representation, more basic than the **Reactive Fabric Element**, as elements are decorated with additional attributed structure over their topological character. We will use the standard mathematical construct of the **Directed Graph** (visualising nodes and arrows) as a topological representation but also stipulate that there are two types of arrows: **one-way** and **bi-way**. We also define a mapping from a **Fabric Element** to its **topological representation** by:

- (i) A single **element** maps to a single **topological node** (but note the third point).
- (ii) A pathway maps to either a **one-way** arrow or a **bi-way** arrow, depending on its pathway directionality (one-way or bi-way).
- (iii) Adjacent topological nodes, with identical in-and-out arrows are merged into a single node representative as they are topologically identical. Also, if there is a terminator node on a single file chain, then the whole operator/socket chain is collapsed to the terminator node.

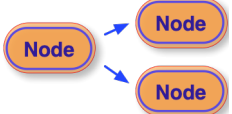
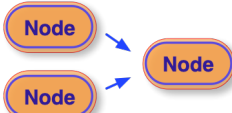
Here are some simple example topological representations and mappings:

Expression	Topology
	
	

Generally, topologies with **circular paths** are not desirable and there are replacement arrangements that can be used to ensure formalised topologies are non-circular. Below is the basic circular pattern with its replacement:

Name	Circular Topology	Non-Circular Equivalent
The Feed-back pattern.		

Also, just for context, here are the two most common topological patterns:

Name	Topology	Description
The <b>Broadcast and Route</b> pattern.		The <b>Broadcast</b> pattern is applicable if the same notification is emitted to both target nodes. If the notification target is selective, then this is a <b>Routing</b> pattern.
The <b>Join</b> pattern.		Joins two notification streams together either by merging the streams, or combining the individual notifications.

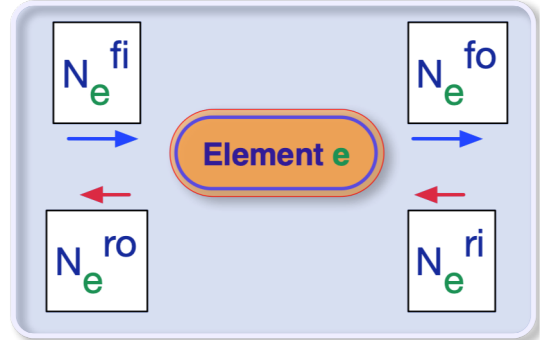
## 8. Mathematical Notation for Element Interactions

### 1. Notification Spaces.

Element processing is effectively a mapping function from the element input **Notification Space** to the element output **Notification Space**.

**Notification spaces** can be discrete (such as enums) or continuous (such as floating point). Typically, they are a composite (i.e. a struct or class object). The notation for the **Notification Space** is depicted here below. The space subscript denotes the **Element Name** (in green), the superscript (in blue) denotes the **direction** (f for forward, r for reverse) and **port** (i for input and o for output).

$$N_{\langle \text{element name} \rangle}^{\langle \text{direction} \rangle \langle \text{port} \rangle}$$



The **Element** state also requires representation, it is indexed over the element operational sequence number:

$$S_{\langle \text{element name} \rangle}(\text{Element op sequence number})$$

### 2. Element Process-Mapping.

The forward element process is expressed as a stateful function of the element input **Forward-Notification Space** and the current **State** of the **Element** to the output **Forward-Notification Space**:

$$f_e^f(N_e^{fi}, s_{e(i)}) \Rightarrow N_e^{fo}$$

The reverse element process is expressed as:

$$f_e^r(N_e^{ri}, s_{e(i)}) \Rightarrow N_e^{ro}$$

If the element does have state, then there is also a mapping on the state from one element op to the next. The state may also change with reverse notifications, so there is a corresponding mapping for that as well. Shown to the right:

$$f_{S(e)}^f(N_e^{fi}, s_{e(i)}) \Rightarrow s_{e(i+1)}$$

$$f_{S(e)}^r(N_e^{ri}, s_{e(i)}) \Rightarrow s_{e(i+1)}$$

If the Element is simple, one-way and functional (without state) these separate mappings collapse into one simple functional (stateless) mapping as shown on the right:

$$f_e^f(N_e^{fi}) \Rightarrow N_e^{fo}$$

The act of chaining elements together is mathematically equivalent to composing the individual function mappings. This composite function maps from the first **Notification Input Space** to the end **Notification Output Space**.

## 9. Scalability and Performance

### 1. Analysing and Handling Performance Bottlenecks.

Message based architectures are generally not considered scalable. **Reactive Fabric** does provide avenues to identify performance bottlenecks and offers mechanisms to handle or mitigate the effects.

Bottleneck identification revolves around analysing pathways with **high throughput** and ascertaining what is driving that traffic. Going further, notification traffic is either driven by a system input pathway or is generated internally, within the **Fabric**. So the driving mechanism must be managed in some way depending on these two cases:

- The driving mechanism is **external** to the Fabric.
- The driving mechanism is **internal** to the Fabric.

The **external** options are:

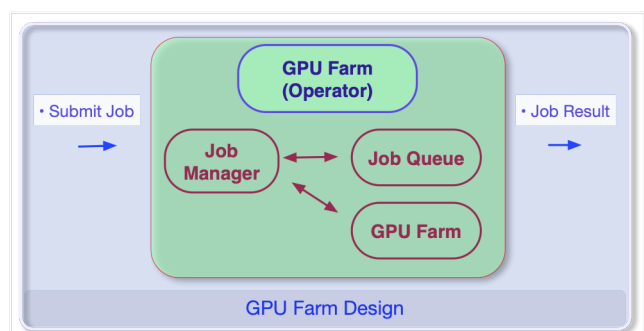
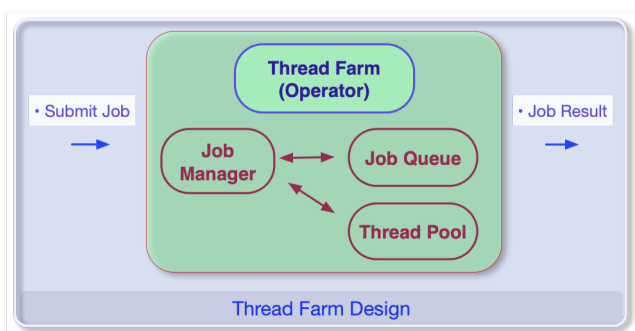
- Design-in an exchange protocol to dynamically auto rate limit the external sources when the local system detects overloading.
- Apply load sharing with separate processing systems (in separate processes) so that notification handling is farmed to a system pool.

The **internal** options are:

- Apply intelligent batching of notifications to reduce messaging and iteration overhead. This is very useful for audio and video processing when using notifications to channel audio/video content.
- Employing **direct data channels** between data-source and data-user elements. Rather than passing notifications along a complicated pathway chain (that do not need to process the those actual notifications). The data-user element emits a request for a **data channel** downstream to the data-source element. The data-source replies back with a **direct data channel** optimized for the situation, which the data-user element then employs for data sending or receiving.
- Optimisation: Ensure the notification is passed by class reference and so does not incur bulk struct copying when being emitted between **Elements**. Also, make sure the source to target element pathway is run synchronously, so as to not traverse thread contexts.
- Perform platform CPU-Thread load balancing (see below).
- Perform platform GPU off-loading (see below).
- Collapse the source to target element chain into a single element and handle the element processing code in the traditional manner (such as optimised assembly).

The **SDK Notification** emission mechanism incurs a minimal CPU overhead as it is merely a function call with the notification passed as a parameter. If the **Notification** memory footprint is large then it is advisable to ensure it is passed as a memory reference i.e. ensure the notification is a class and not a struct to minimise copying between function calls. Performance analysers can also be used to localise the cause of performance bottlenecks.

Bottlenecks are also possible if all available threads/CPU's are not being properly utilised. Here platform CPU and GPU resources can be separately modelled as a **Fabric Service** or **Operator**. That resource element is made CPU/GPU aware and received notifications are then internally farmed out in a balanced manner to those resources. The **Fabric** then routes intensive notifications to these services that guarantee balanced processing. Their results can also be queued and batched to minimise **Service/Operator** element emission rates.



## 📄 10. Code Examples

### 1. Introduction to Code Samples.

Before sitting direct code examples, some background information is first required. The various code samples given here in this document are authored in the **Swift 5** language.

### 2. Background: Element and Notification Representation.

All **SDK** definitions are organised into a namespace hierarchy with the base designation: "**Rf**". Each design element in the design space is mapped to a single generic class in the code space. In these classes, the generic parameter **InItem** is used to denote the reception notification data-type and **OutItem** to denote the emission notification data-type. Here are the declarations for the abstract Element classes that support single-way notification:

```
public class ASource<InItem>           // The Source Element Abstract Class
public class AOperator<InItem, OutItem> // The Operator Element Abstract Class
public class ACollector<OutItem>       // The Collector Element Abstract Class
```

Element notifications are performed by exercising the **notify()** method on the Element-class while passing the **Notification-data** and its **Virtual-timestamp** (of type **Rf.VTime**) as parameters. Notifications come in two varieties: **Item** notifications, as used in Fabric Design and **Control** notifications (used by the **SDK** to coordinate Element operation). Here are their declarations:

```
public func notify(item: InItem, vTime: Rf.VTime?) // Item notify method
public func notify(control: IRfControl, vTime: Rf.VTime?) // Control notify method
```

### 3. Background: Defining Notification Handling within Elements.

The behaviour of an element is described by defining how the element responds to received Notifications. This behaviour is stipulated by the client by defining two closures, one for each Notification category:

```
public var onItem : (InItem, Rf.VTime?) -> Void // Handle item notification
public var onControl: (IRfControl, Rf.VTime?) -> Void // Handle control notification
```

Within these client defined closures, notifications can be emitted to the next downstream element by exercising the notify method on the **producer** property of the Element-class. For example:

```
producer.notify(item: item, vTime: vTime)
```

### 4. Background: Initiating and Terminating Evaluation.

Expression/Element evaluation is initiated through the Element-class method: **beginEval()** and correspondingly, terminated with **endEval()**. Here are their declarations:

```
public func beginEval(_ evalType : eEvalType, vTime: Rf.VTime?) // Begin Evaluation
public func endEval(vTime: Rf.VTime?) // End Evaluation
```

## 📍 10. Code Examples

### 5. A Simple Operator Example: The **Observe** Operator

Below is an example of one of the simplest of operators, the **Observe** operator which runs a given closure: **onItemFunc**() on each received item notification (and then emits the notification). In the code below, the **SDK** factory is used to create the operator instance and then the customization of its notification handlers is performed. In this particular operator, only the **onItem** handler is defined and the **onControl** handler takes its default action:

```
extension Rf.Operators
{
    public static func observe<Item>(_ traceID: Rf.TraceID,
                                     item onItemFunc: @escaping (Item, Rf.VTime?) -> Void
                                     ) -> Rf.AOperator<Item, Item>
    {
        let observeOperator: Rf.AOperator<Item, Item> = RfSDK.factory.Operator(traceID)

        observeOperator.consumer.onItem = { [weak observeOperator] (item: Item, vTime: Rf.VTime?) in

            guard let strongOperator = observeOperator else { return }

            // Pass the notification to the given closure.
            onItemFunc(item, vTime)

            // Emit the notification.
            strongOperator.producer.notify(item: item, vTime: vTime)
        }

        return observeOperator
    }
}
```

Normally, the operator logic is defined as a static in the **Rf.Operators** namespace, and then an extension in the abstract Element class (**Rf.AElement**) is introduced to support composition of the element. Here is the extension for the observe operator:

```
extension Rf.AElement
{
    @discardableResult public func observe(_ traceID: Rf.TraceID = Rf.TraceID("observeItem"),
                                           item onItemFunc: @escaping (OutItem, Rf.VTime?) -> Void
                                           ) -> Rf.AOperator<OutItem, OutItem>
    {
        let observeOperator = Rf.Operators.observe(traceID, item: onItemFunc)
        // Compose the previous operator (i.e. self) with the observeOperator.
        compose(observeOperator)
        return observeOperator
    }
}
```

The operator can then be exercised in the following manner and the **onItemFunc** closure here is defined so as to print the received notification items. The expression in the code below is evaluated asynchronously:

```
// Note: definitions for mySource and myOperator are not defined here.

func onItemFunc(_ item : Int, _ vTime: Rf.VTime? = nil) { print(item) }

let mySource = Rf.Sources.mySource<Int>()

mySource.observe(item: onItemFunc).myOperator().beginEval(.eAsync(nil))
```

## 📌 10. Code Examples (cont)

### 6. Operator Example Handling Both Item and Control Notifications.

Below is a **support** operator called **onNotify** which takes a given general **notifyFunc()** closure and funnels all notifications (both item and control) through to that closure. The operator is used in the next time-based operator example. This operator additionally tracks the item sequence index and supplies that to **notifyFunc()** together with the item notification. It also handles Fabric tool installation which is used to disseminate tools such as **schedulers** to the Element.

```
extension Rf.Operators
{
    public static func onNotify<InItem, OutItem>(
        _ traceID : Rf.TraceID,
        _ notifyFunc: @escaping (Rf.eOperatorNotification<InItem>, Rf.AElement<InItem, OutItem>) -> Void
    ) -> Rf.AOperator<InItem, OutItem>
{
    typealias eFabricNotify = Rf.FabricScope.eNotify
    typealias eEvalNotify = Rf.EvalScope.eNotify

    let onNotifyOperator : Rf.AOperator<InItem, OutItem> = RfSDK.factory.Operator(traceID)

    var currentIndex = 0

    onNotifyOperator.consumer.onItem = { [weak onNotifyOperator] (item: InItem, vTime: Rf.VTime?) in
        guard let strongOperator = onNotifyOperator else { return }

        notifyFunc(.eItem(currentIndex, item, vTime), strongOperator)

        currentIndex += 1
    }

    onNotifyOperator.consumer.onControl = { [weak onNotifyOperator] (control, vTime) in
        guard let strongOperator = onNotifyOperator else { return }

        switch control
        {
            case eEvalNotify.eEvalBegin(let evalType):
                if let keys = strongOperator.tools?.keys
                {
                    for name in keys
                    {
                        notifyFunc(.eFabricTool(name, strongOperator), strongOperator)
                    }
                }

                notifyFunc(.eEvalControl(.eEvalBegin(evalType), vTime), strongOperator)
            case eEvalNotify.eEvalRestart(let evalType):
                currentIndex = 0

                notifyFunc(.eEvalControl(.eEvalRestart(evalType), vTime), strongOperator)
            case eEvalNotify.eEvalEnd(let error):
                notifyFunc(.eEvalControl(.eEvalEnd(error), vTime), strongOperator)
            case eFabricNotify.eInstallTool:
                strongOperator.producer.notify(control: control, vTime: vTime)
            default:
                strongOperator.producer.notify(control: control, vTime: vTime)
        }
    }

    return onNotifyOperator
}
}
```

## 📌 10. Code Examples (cont)

### 7. Temporal Operator Example: The throttle Operator.

The **throttle** operator is defined below to propagate item notifications at a given rate as specified by the **duration** parameter. A fabric scheduler is used to control the temporal window that disallows item notification propagation. The scheduler is disseminated by the fabric to the element before evaluation and passed as the **eFabricTool** input notification:

```
public static func throttle<Item>(_ duration : TimeInterval, settings : Rf.SchedulerSettings) -> Rf.AOperator<Item, Item>
{
    typealias eEvalNotify = Rf.EvalScope.eNotify

    let traceID                = Rf.TraceID("throttle")
    var scheduleTool : Rf.Tools.Schedule? = nil                // The fabric scheduler.
    var allowItemsToPass      = true                          // Indicator of whether to pass item notifications.

    let throttleOperator : Rf.AOperator<Item, Item> = Rf.Operators.onNotify(traceID, { (inputNotification, element) in
        switch inputNotification
        {
            case .eFabricTool(let (name, tools)):                // On a Scheduler Fabric Tool install request.
                if name == "scheduler", let tool = tools.getScheduleTool(traceID)
                {
                    scheduleTool = tool                        // Set the scheduler.
                }

            case .eEvalControl(.eEvalBegin(let evalType), (let vTime)): // On a begin evaluation Control notification.
                guard let scheduleTool = scheduleTool, let evalQueue = evalType.evalQueue else
                {
                    RfSDK.assertionFailure("\(traceID): No scheduler or evalQueue available"); return
                }

                evalQueue.dispatch(sync: { allowItemsToPass = true }) // Set allowItemsToPass in the evaluation queue thread.

                scheduleTool.evalQueue = evalQueue
                scheduleTool.subscribe(settings: settings)           // Subscribe to the scheduleTool.

                // Propagate the Eval Begin event.
                element.producer.notify(control: eEvalNotify.eEvalBegin(evalType), vTime: vTime)

            case .eItem(let (_, item, vTime)):                // On an Item notification.
                // Selectively propagate item notifications given by allowItemsToPass.
                if allowItemsToPass
                {
                    allowItemsToPass = false

                    // Schedule a throttle time window, depending on the type of time reference we have for the current time.
                    if let vTime = vTime
                    {
                        // A virtual time reference is available, use it to schedule the next processing event.
                        scheduleTool!.schedule(atVTime: duration + vTime, action: { (time) in
                            allowItemsToPass = true
                        })
                    }
                    else
                    {
                        // Use a real time reference instead to schedule the next processing event.
                        scheduleTool!.schedule(atTime: Date(timeIntervalSinceNow: duration), action: { (time) in
                            allowItemsToPass = true
                        })
                    }

                    element.producer.notify(item: item, vTime: vTime)
                }

            case .eEvalControl(let (evalEvent, vTime)):        // On any other control notification.
                if let scheduleTool = scheduleTool
                {
                    scheduleTool.unsubscribe(vTime: vTime)    // Unsubscribe from the scheduleTool.
                }

                element.producer.notify(control: evalEvent, vTime: vTime) // Propagate the event.
        }
    })

    return throttleOperator
}
```



## 📍 11. Technology Side Aspects

### 1. Hybrid Systems.

**Reactive Fabric** permits a great deal of scope and flexibility in what it is employed for. It does not need to be utilised on all system aspects, it can be used for specific purposes and targeted application. For example, it can be just used on the **Macro** behaviour level, performing high-level management. It can be used to funnel requests and data to and from **non-Reactive Fabric** processing units.

These alternate **non-Reactive Fabric** processing units may include:

- **AI** processing nodes, (i.e. learning systems).
- **Legacy code** that is not intended to be changed or is currently in the process of being refactored from a traditional model to **Reactive Fabric**.
- **Cross Platform IP** that is not intended to be ported to **Reactive Fabric** but instead to be wrapped by it.
- **Third Party Libraries** for which source is not available.
- **Processing farms** (GPU/CPU/Remote RPC/Render).

**Reactive Fabric** is also a natural medium for streaming data and orchestrating high level management. This makes it a prime complementary technology for integration with learning systems.

### 2. Conversion of Legacy Systems.

In scenarios of legacy system conversion, not all of the legacy code may require to be converted.

There are also many conversion developmental paths:

- The general recommended method is to convert piecemeal, taking sections with clear boundaries. First identifying internal services and converting those to **Reactive Fabric** services. Then identifying IO pieces converting those and finally filling in the appropriate holes.
- Convert in top down fashion, converting **Macro** behaviour first, so that it orchestrates existing **Micro** behaviour and then in piecemeal, convert the appropriate underlying legacy code.
- Convert by addressing areas of concern first (such as performance concerns).

During conversion, there may be a number of designs produced, one being for an ideal design (for a whole new system) and others that constitute more workable designs that mate with the existing system contour-boundaries to be more advantageous for the conversion process.

### 3. Future Possibilities for Reactive Fabric.

**Reactive Fabric** may in the future, have more tools such as:

- Visual tools to input designs, in the same way that UML employs visual input tools.
- Auto-generating implementation skeletons from designs (which are then manually filled in).

**Reactive Fabric** is a medium in which to express design and as **AI** plays more of a role in the generation of systems, **Reactive Fabric** designs could be an appropriate medium in which to express human comprehensible **AI** generated systems.

### 4. Current Platform Support.

Currently, the **Reactive Fabric SDK** supports the **Swift 5** language. The roadmap does include the **Kotlin** language. (The Linux platform is also supported by the Swift language).

## 🌀 Appendix 1. Reactive Fabric Origins, History and Comparisons

### 1. History of the Reactive Fabric Concept.

**Reactive Fabric** orientates and synchronises the individual design-development processes to work together through pure design, which even includes the design of the development process itself. It amplifies the benefits of its individual facets to become more than the sum of its parts.

**Reactive Fabric**, stands as a confluence of a number of previous software technologies and influences including: **Reactive Extensions**, **Micro-Processes**, **Functional Programming**, the more distant work of the **Haskell Language**, **Software Patterns**, **Sequential composition**, **UML** design and the concept of **Event Streams**.

With respect to these software technologies:

🌀 The **Haskell** language provided the abstraction where you don't merely define **static** behaviour but dynamically **generate** the target behaviour. This decouples the temporal epochs of when behaviour is **defined** from when it is **executed** and makes that behaviour a **first class citizen**, so that it can be handed over to a distant execution point. So basically, rather than writing a function to do something, we define a function which will construct a function to do that something and then pass it on to where it is to be executed. This also supports provisioning of resources at the time that it is required (execution time rather than at definition) and the provision for post customisation of the generated behaviour. Haskell also provided the concept of **functional composition** (which also goes by other names of sequential composition and fluent programming style). It also introduced the concept of the **functor** (where function behaviour is transformed through mapping functions).

🌀 **Reactive Extensions** came much later and drew upon Haskell concepts. It melded the ideas of the **Event Stream** with the **Observe/Observable Pattern** to be what we know it as today. Reactive Extensions was another software epoch, but it was developed as a strict library of **Observables** and **Observers** (take, skip, interval, zip, etc). As a bulky library, it was never abstracted and condensed to a concise set of extendable principles. It was scoped as solely at the implementation level and as such does not engage in the design process. Reactive Extensions also hides its internal operation, so it does not inherently permit the developer to participate together with its operation. The framework is more of a remote mechanism which the developer engages rather than participating with.

🌀 **Micro-Processes**: where Reactive Extensions provides a well defined conceptual framework and code interface, the Micro-Process concept has been too abstract, with insufficient support to define how messaging is performed and with no patterning or classification of the different roles and arrangements of the individual micro processes. (Adding more context, various groups have gone on to put forward their own messaging protocol and serialisation schemes). The Micro-process concept was innovative, but it lacked implementation support and design backing.

🌀 **UML Design** was a blueprint tool, more of a representation of the implementation rather than a description of the pure design. It was too low level (suffered from over detail) and overly coupled to the target (class-object) language. It did not cater for levels of detail and as such, quickly became unwieldy. Also, its target audience is limited to software specialists.

🌀 **Event Streams** is a core implementation only concept and by itself, does not handle complex asynchronous scenarios.

Using the metrics of **Domain** (i.e. design and implementation) and **Scope of Behaviour** (Macro, Mid and Micro) to classify the applicability of software technologies, here is a table classifying the coverage of the respective technologies we have just discussed:

Name	Applicable Domain	Scope of Behaviour
<b>Reactive Extensions</b>	More Implementation and less Design	Micro, reaching towards Mid
<b>Micro-Processes</b>	More Design and less Implementation.	Macro
<b>Event Streams</b>	Implementation only	Micro only
<b>Reactive Fabric</b>	Implementation and Design balanced.	Macro to Micro

## 🌀 More Information

---

### 1. For More Information.

Currently Reactive Fabric supports the Swift language (which does support the linux platform) but is road mapped for other languages such as Kotlin.

#### 🌀 Reactive Fabric White Papers and Briefs:

- The Reactive Fabric White Paper, in three parts:

- [Reactive Fabric White Paper Part 1 of 3.pdf](#)
- [Reactive Fabric White Paper Part 2 of 3.pdf](#)
- [Reactive Fabric White Paper Part 3 of 3.pdf](#)

- For Desktop and Mobile Apps:

- [Reactive Fabric For Desktop and Mobile Product Application.pdf](#)

- For IOT:

- [Reactive Fabric For IOT Product Application.pdf](#)

#### 🌀 Website pages:

- Originware site: [originware.com](http://originware.com)
- The Reactive Fabric web page: [originware.com/reactivefabric.html](http://originware.com/reactivefabric.html)

### 2. Reactive Fabric Samples.

Apps designed on Reactive Fabric principles:

- 🌀 The **Dronenaut** iOS App on the App Store [The Dronenaut App](#)  
(employs the older **Reactive Patterns SDK**)

- 🌀 The Reactive Fabric Sample App: [Reactive Fabric Sample Kit For iOS](#)  
(employs the **Reactive Fabric SDK**)

### 3. Contact.

Please direct questions and requests for more information to:

Terry Stillone ([terry@originware.com](mailto:terry@originware.com))