Reactive Fabric Technology

SDK For iOS, OSX (and coming for Linux)

White paper

September 2018

Originware
Techology

Author: Terry Stillone (*terry@originware.com*)

Web: *www.originware.com*

Version: 1.0

1. Introduction

1. You, the Audience

Whatever your background, this white paper is addressed to you .... but, perhaps not all of its content is relevant for you.

To support various reader technical levels, this document is structured to gradually progress from a more generic, conceptual description of **Reactive Fabric** technology, to a more software architectural stance and then finally engage in detail with code examples. So you may want to begin reading and then fall off as the conversation loses relevance for your background.

Just before we start, I will formally define the domain of the technology:

Reactive Fabric *is a software technology that covers software engineering aspects of*:

- The **Software Design process**, modelling software operation with processing **Elements** and **Notifications**.
- The **visualisation of the Software Design** through **Element** visual models in various levels of detail (**LOD**).

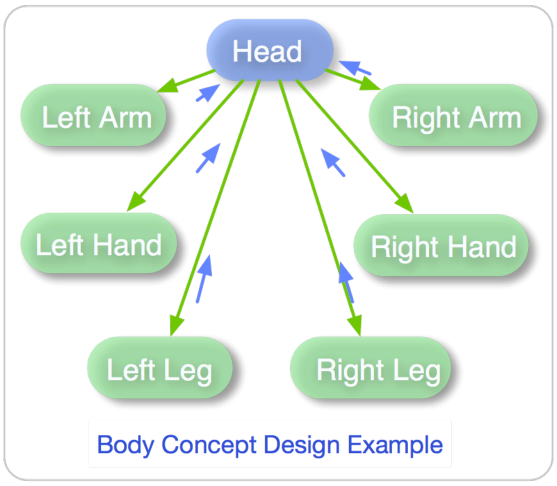
*The **Reactive Fabric** process engages*:

- **Collaboration** of the design process.
- **Thought based prototyping** in the **Reactive Fabric Design Space** with less emphasis on software prototyping.
- **Sharing** and **Comprehension** of the design models with all associated parties at the appropriate **LOD.**

The inclusive **Reactive Fabric SDK** written in the **Swift** language provides a direct mapping of design model **Elements** and **Notifications** to software classes and manages the evaluation of client defined fabrics.

2. Reactive Fabric Technology in Conceptual Terms.

Lets take a simple example, something we are all familiar with and treat it as a system design. Lets treat the operation of the human body as a **system** and create a **concept system design**. As a simple, high level mechanical system, the human body looks something like the diagram below. The elements (head, arms, hands and legs) form the functional elements of the design. The green forward arrows indicate direct control interaction in the form of notifications (messages) between elements. The smaller secondary blue (back directing) arrows indicate back reply notifications resulting from previous forward notifications.



Body Concept Design Example

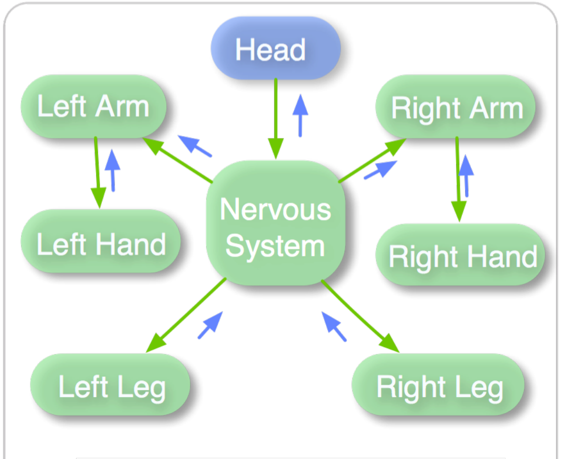
In our design, the head element notifies appendage elements of direct muscle requests and in turn the head receives back replies indicating muscle response (and possibly appendage positional telemetry as well). Each element has its own set of notification requests and the elements being messaged must recognise those requests and respond with appropriate replies. As this is an initial concept design, the specific types of notifications are not stated here. Only their role or messaging intent is given at this design scope level.

This design follows a "***Centralised Control Pattern"*** as the head controls everything and the secondary appendage elements merely respond. In this case the notification traffic volume is high as the forward and back replies perform a **Feedback Pattern** which only diminishes when the appendage has reached its target position and stops.

1. Reactive Fabric Technology.

Given that initial system how about we perform a small redesign shifting away from a high bandwidth notification design (using direct muscle control commands) to one using more high level directive notifications which are also routable. The notifications in this version indicate motion intent and the target appendage.

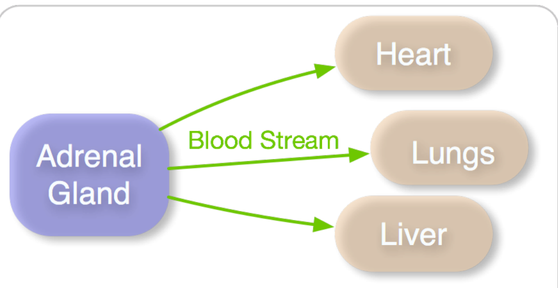
In this modified version (to the right), the head issues requests in the form of intent and addresses notifications to target elements using the **Nervous System** element as a routing pattern. The head for example may issue a raise left arm with a target of the left arm. The notification will contain the "raise arm" request with the target of "left arm" to the Nervous System element which will look at the target entity at and pass the notification to the **Left Arm**. The arm then manages its motion and replies back with positional information when the motion is complete. The notifications in this design merely message high level control (intent) and so the general notification bandwidth is markedly reduced.



Decentralised Body Design Example

In this design, routing notifications through the **Nervous System** gives us some advantages we can make use of. First the nervous system is the activity gateway for both commands and replies. So it is a natural access point for monitoring whole system activity. Additionally, we can use the **Nervous System** element to inject notifications into the whole system so as to simulate activity. With this, if we wanted to test the system we could inject the desired commands into the **Nervous System** element and then monitor the direct response of the appendages and what is passed onto the head.

In the previous body design, we were describing a "**macro system behaviour design model"**. We can also come up with medium level designs. Here on the right is a simple concept design for the adrenal glands. Notice this is a one way messaging model as opposed to the previous bi-directional model. Here the Adrenal glands message through the blood stream pathway and messaging is performed by a hormone molecule rather than an electrical impulse (as in the previous model).



Adrenal Gland Concept Design Example

As this is a concept design, it has been presented as a standalone model where in reality the glands are directed by the nervous system (element). That model relationship would be depicted later, in more refined design iterations where the emphasis is more on detail rather than concept. In the concept design we want to convey the architectural level of operation and not include too much detail and not make assumptions on how it must function (leaving those decisions for the subsequent functional design iterations).

We could also come up with micro-level designs for cell operation. Where in actual fact cell operation is extremely complex, we don't need to put all that into our models. We only need to put into our designs the operations that are relevant for the project and relevant for the current design phase. So we only design for what is required.

2. Participation in the Reactive Fabric Design Process.

Reactive Fabric engages the whole team in the design process, from product management, project management, software design, software development, testing, documentation all the way to marketing. This engagement is made possible by making the design models and the design process available to all and most importantly: **available at their own level of engagement** through model LOD (Level Of Detail).

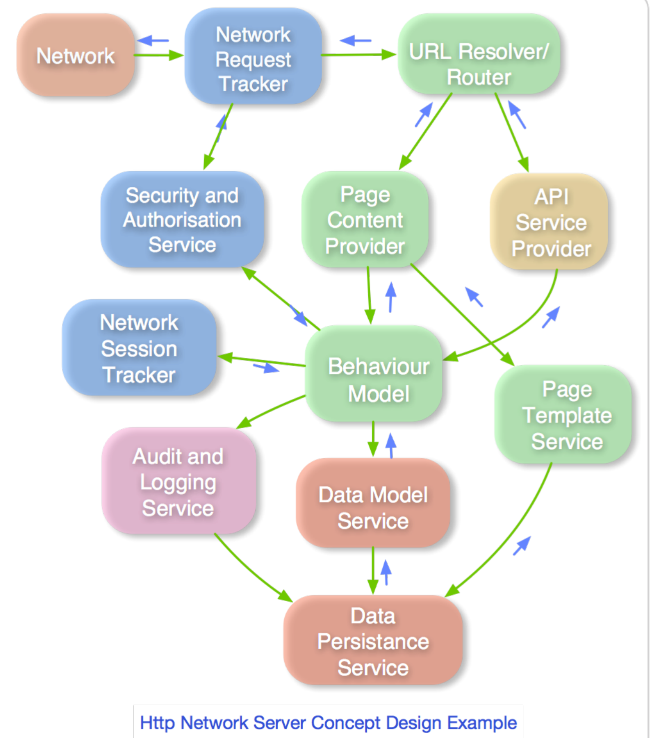
Lets look at some example design iteration scenarios:

- In the initial design meeting, everyone joins in to sketch out the concept design models. The models naturally define functional boundaries, which translates to how teams are formed and their individual team sizes. Those teams then individually over time work on their given concept models to detail and mature for a first iteration model and they present those back in a subsequent group meeting. Model feedback is given during the iteration design meeting and design constraints are defined relevant for the time. An initial iteration 1 implementation is constructed from the iteration 1 design model. The implementation is tested which then feeds back into the iteration 2 group design meeting and so on.
- Product management and marketing comes in to look at macro design level models and give input on additional high level features and capability. The software development team then pushes back on some of those ideas by showing in model terms how much extra work those features are to attain. Understanding that, product and marketing then modifies their ideas to come up with suggestions that better trade off between work and feature benefit.
- Over time, models are expanded, matured and refined (together with their implementations). If the group is following a classical developmental path, then towards the final iterations the designs converge and change by smaller increments. Alternatively, the design process can also bring up new possibilities which after collective assessment may be incorporated. The point is that through collective envisioning of models and collective participation, more opportunities are realised.

Through the model, participants have both a visual model and a vocabulary. Engaging them to discuss, give insight, suggest, comprehend, explain and trade off.

Design models can be platform vertical as they can cover from server level operation, through to large application, to mobile app and SOC (embedded) systems and platforms. Modelling brings project design unification and various teams can reference other team models to support their own.

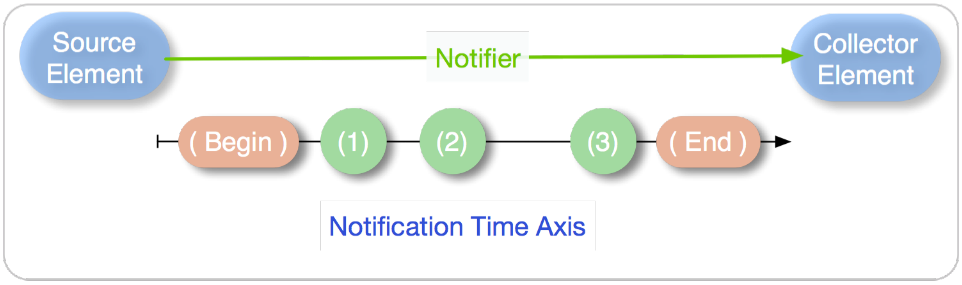
To give a better idea of a practical design model, the example to the right is a Rest Web-Server design. The system serves HTML pages. HTTP requests come through the **Network** element, which are passed through a **Network Session Tracker** for validation, then to the **URL Router** element to perform routing to the appropriate content provider. For page content, the **Page Content Provider** generates the HTML page employing a **Page Template Service**. The generated page is then passed back through the same path to the **Network** element for delivery.



Http Network Server Concept Design Example

1. The Concept Of The "Notification"

The "**Notification"** is an analogy of the *electrical impulse*, the *chemical messenger* or the *network data packet* in the computer science domain. In abstract terms, it is a transportable data entity (a first class object) which requires a transport path from one **Element** to another. It encapsulates the information that is to be messaged between particular elements over time and as such, collectively constitutes a data stream of data events (with their data payload) notifying the target Element.



This is typified diagrammatically here with a source emitting a sequence of notifications to a collector over time. These notifications include both control events (in brown) and item events (in green with integer data payloads).

Side note: The diagram above depicting notifications as balls on a timeline has a formal name of:

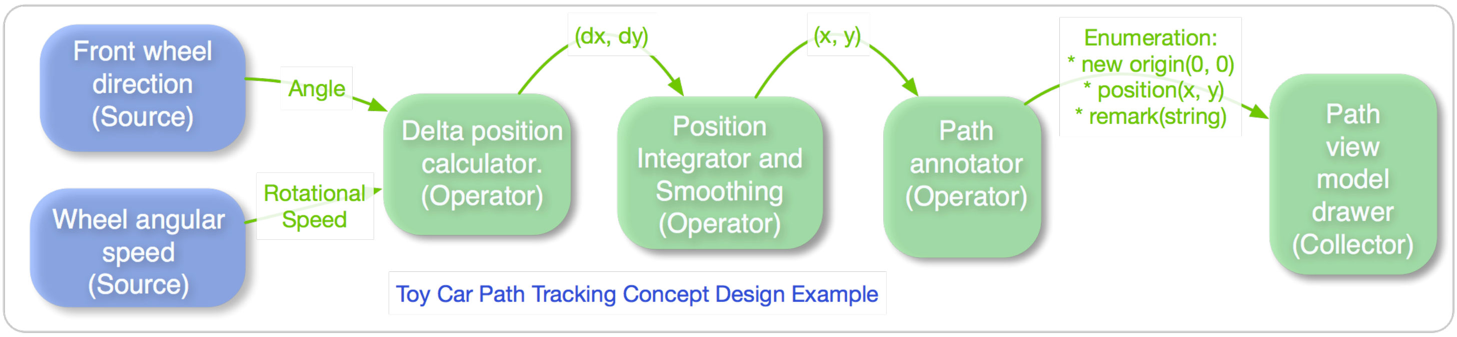
"**Marble Diagrams**"

For a complete set of Reactive Extension marble diagrams, see:

http://rxmarbles.com/

As **Source** elements produce notifications, **Collector** elements consume, **Operator** elements perform computations on notifications by consuming and producing. Operators use their input notification to calculate a corresponding output notification. Thus a system's behaviour and state is described by notifications flowing through the system in the same way that a physical organic system changes its state as a response to stimuli which ultimately forms its whole behaviour.

To flesh out the interaction of elements and notifications, lets look at a direct example. Say we have a toy remote control vehicle and our Reactive Fabric app receives telemetry back from the remote toy. The telemetry includes the toy's wheel rotation speed (in revolutions per sec) and front wheel direction (in degrees). Our App is required to perform realtime analysis, to compute the path of the vehicle and draw it on the device's screen. Our design describes the receiving telemetry as two source elements, one for wheel rotational speed and one for wheel direction. We utilise an operator to perform a convolution of the two source element notifications to calculate the incremental positional change (the velocity vector) and then feed that into an integration operator to calculate the position of the toy. Here is what the practical design looks like:


Front wheel direction (Source)
Angle
Wheel angular speed (Source)
Rotational Speed
Delta position calculator. (Operator)
(dx, dy)
Position Integrator and Smoothing (Operator)
(x, y)
Path annotator (Operator)
Enumeration:
* new origin(0, 0)
* position(x, y)
* remark(string)
Path view model drawer (Collector)
Toy Car Path Tracking Concept Design Example

Going into more detail, when we are drawing the toy motion, we also want a smooth path on the screen, so lets add smoothing capability to the **Position Integrator** operator. We also need to put some text up on the screen to describe the speed of the toy, (i.e. stopped or in motion with a speed level). So lets add an **Annotator** operator which takes the position from the **Position Integrator** and analyses the position at various time increments to emit a resultant enumeration indicating either of: 1. An indicator of the start of a new path (implying the previous path should be cleared), 2. An update to the current toy position (when the position changes) relative to the start point or 3. The toy motion speed (when in motion).

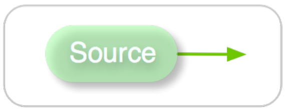
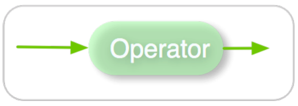
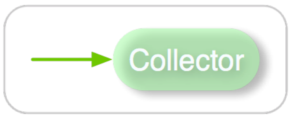
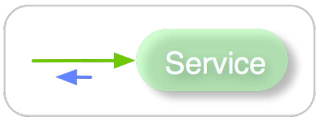
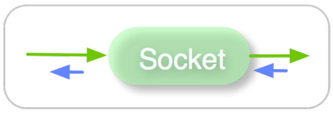
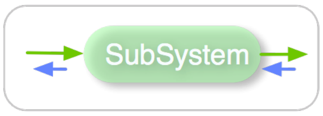
All the results of the **Annotator** then feeds into the element that becomes the presentation **View Model** which draws the toys path on the screen together with the speed indication text.

1. Reactive Fabric Patterns

Reactive Fabric abstracts common Element roles into a set of **Reactive Fabric Patterns**. These patterns describe the elements general processing role. These role variations are also intrinsically reflected in variations of their notification pathway topology (i.e. whether they have inputs, outputs, replies, etc). So the list below sites the patterns role together with their topology variance.

These patterns also form the base vocabulary for verbally describing and for the discussion of models. (e.g.: "The text-input **Source** feeds into the app-logic **Subsystem** which turn passes data requests to the HTTP **Service**")

(Legend: Green arrows identify notification pathways and blue arrows denote back reply paths)

Pattern	Description	Topology
The Source Pattern	Soley generates Rf.Notifications (i.e. does not receive notifications).	
The Operator Pattern	Operates on input Rf.Notifications to derive output Rf.Notifications.	
The Collector Pattern	Collects Rf.Notification and makes them available for consumption.	
The Service Pattern	Provides a notification Service, accepting request notifications and replying back via callback closures (these closures are defined in the original request). The callback is denoted in the diagram as a blue back notification arrow.	
The Socket Pattern	A hybrid of the Service and Producer patterns. Sockets accept notification requests, similar to the Service Pattern but also generates notifications.	
The SubSystem Pattern	A SubSystem manages a sub-collective of Elements. SubSystem Rf.Notification inputs and outputs are made available to the sub-collective for processing as the SubSystem element in itself does no processing other than routing input and outputs.	

Within a typical system design, particular elements by virtue of their placement within the topology of the Fabric will form particular roles. For example, fabric boundary elements will end up inputing or outputting information. So inputs will be **Sources** and outputs tend to be **Collectors**. Internally, simple processing elements are typically **Operators** while complex processing will tend to be organised into **SubSystems**. Shared facilities within the Fabric will tend to be **Service** elements**.** Complex pipelines such as network protocol stacks will normally be designed as a sequentially composed **Sockets**.

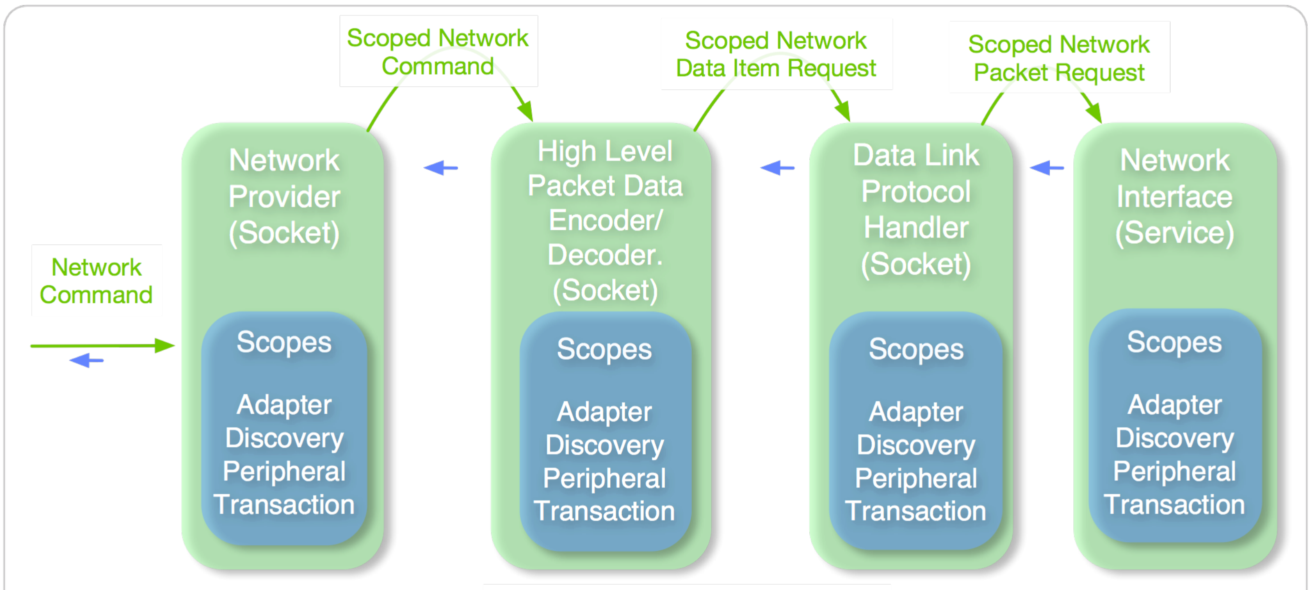
2. Element Scoping

Reactive Fabric also supports the capability of Element **Scoping**. Element-Scoping is required when the whole system operates in a number of separate modes or states and the element processing must handle and process in relation to those modes. This means that Element logic may be different for individual scopes, it may also mean that the elements accept or emit varying notification data-types depending on the current scope.

Protocol Stacks are a good application match for scoping where the elements are required to function under various network protocol operational phases. A bluetooth protocol for example has phases which include:

• Adaptor Initialisation phase:	(with actions: open/close adapter),
• (Peripheral) Discovery phase:	(discover peripherals)
• (Peripheral) Connection phase:	(connect/disconnect to peripheral, discover peripherals services)
• (Peripheral) Transaction phase:	(read data, write data)

If we map each bluetooth layer to a processing element (**Socket**) where each is scoped for these modes, the design looks something like:


Scoped Network Command
Scoped Network Data Item Request
Scoped Network Packet Request
Network Command
Network Provider (Socket)
High Level Packet Data Encoder/ Decoder. (Socket)
Data Link Protocol Handler (Socket)
Network Interface (Service)
Scopes
Adapter
Discovery
Peripheral
Transaction

BLE Protocol Stack Design Example

In more depth: the **Network Provider Element** provides the Network Service to the whole system. It accepts requests such as *Open Adapter*, *Discover Peripherals*, *Connect to Peripheral*, *Get/Put data*, etc depending on the current scope. Those issued commands are propagated through the protocol element pipeline (representing the various protocol layers) and when a reply to a command is generated (normally at the **Network Interface** element) it is propagated back to the **Network Provider** and then back to the issuing client.

Each element, performs a distinct role in the protocol stack and each operates in relation to their current scope. So in the Adapter scope level case, the command is merely passed through to an element that can perform the action (that is the final **Network Interface** element). The **Network Interface** element acts a device driver (interfacing to the OS), performing the adapter opening or closing (through the given OS interface) and then returns a result as a reply which is back propagated.

A more general applicable example of scoping is to manage Application modes where the application must bootstrap various levels of operation before performing actual application logic. A typical app will have applicable modes of: *App Initialisation*, *Security Handling*, *OS Service initialisation*, *Presentation System Initialisation* and the execution of the *Application logic*.

3. Synchronisation and Evaluation Ordering.

Reactive Fabric supports a concurrency model similar to Apples' **Grand Central Dispatch**. Evaluation is performed by and run within what is termed an "**Evaluation Queue"**. An **Evaluation Queue** accepts a closure (to be executed) and runs the closure if the dedicated queue thread is available, (running it in that thread), otherwise it queues the execution request for when thread availability is reestablished. The **Evaluation Queue** is an enhanced version of the Apple **Dispatch Queue**, enhanced in that it contains special anti-deadlocking support.

In support of synchronisation, each **Fabric** instance is assigned a serial **Evaluation Queue** and then its evaluation (and inner notifications) are performed in the dedicated (serial) evaluation queue. This ensures fabric operations are properly ordered and fully synchronised. The **Evaluation Queue** scheme places threading issues very much in the background and for the developer it only requires attention under special circumstances such as joining fabrics or offloading intensive operations.

Element co-operations such as timer actions are run in their own timer queue but the results are handed over to the target element in the fabric **Evaluation Queue** to guarantee fabric synchronisation.

If the fabric has heavy processing operations to the point execution off loading is required, additional **Evaluation Queues** can be created and used.

4. Modes of Fabric Evaluation.

Fabric evaluation can be performed in a number of executional modalities to suit the concurrency requirements of the evaluation. These modes are:

- **Asynchronous**, running in the fabric EvalQueue with the caller not being blocked.
- **Synchronous**, running in the fabric EvalQueue with the caller blocked until evaluation completion.
- **Inline**, where the calling Evaluation Queue (or native Dispatch Queue) is used to perform the evaluation. The caller is blocked in the sense it is used for evaluation. This modes is designed to reduce thread overhead.

5. Scheduling Internal Element Operations: The Scheduler Tick Service.

The **Fabric** has the capability to deploy resources to its encompassing elements to coordinate collective evaluation. This resource dissemination is carried out before the initiation of evaluation through a series of Fabric notifications. Once deployed, these resources can be utilised during evaluation. Schedulers for example can be disseminated to elements for use by element timers. These schedulers support a virtual time stream that counters problems that may arise from system timer coalescing.

[*Background on timer coalescing : Modern systems employ timer coalescing to perturb system timer events so as to align and reduce system energy demands. This severely impacts on timer ordering in a complex system as no longer does a timer trigger strictly correspond to the system clock and various sub-systems can no longer agree on ordered timed events*].

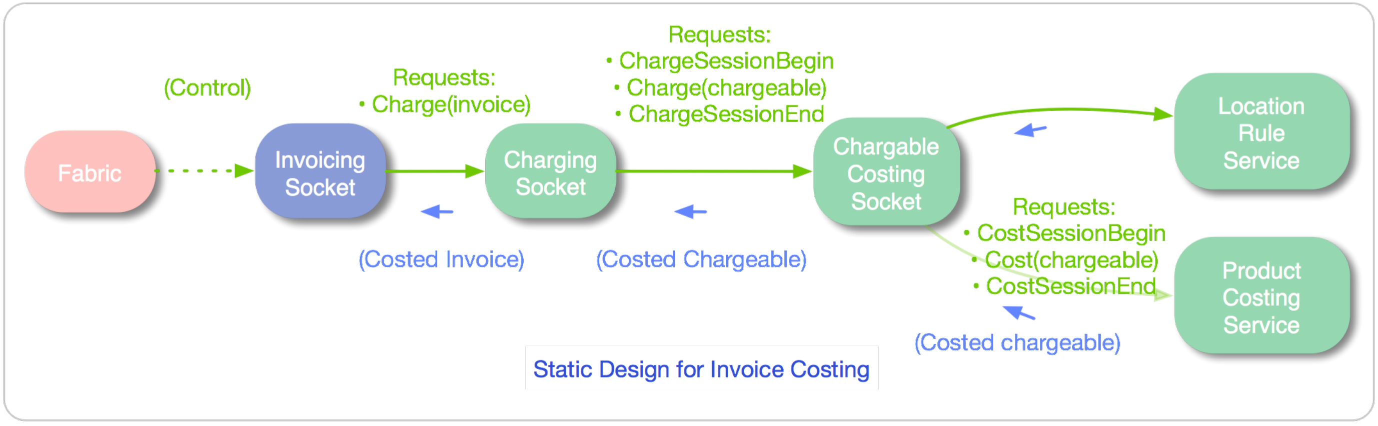
The Reactive Fabric virtual time schedulers ensure their time values are in order and are comparable across separate elements and fabrics. Notifications can be optionally timestamped with their associated virtual time so that element processing has a time reference.

6. Adaptive Behaviour: Element Behaviour Injection and Adaption.

In traditional design, Elements have their logic hardwired in their element handling code. This static design works for non-adaptive requirements but what about scenarios where evaluation must adapt for situational changes? If the case demands that fabric topology remains invariant to situational changes, but the individual element behaviour must change, then why not get the element behaviour to adapt or preconfigure to the situation? By using Fabric events (raised before evaluation initiates) configuration and customised behaviour can be passed from the fabric to the element and then executed in the evaluation cycle.

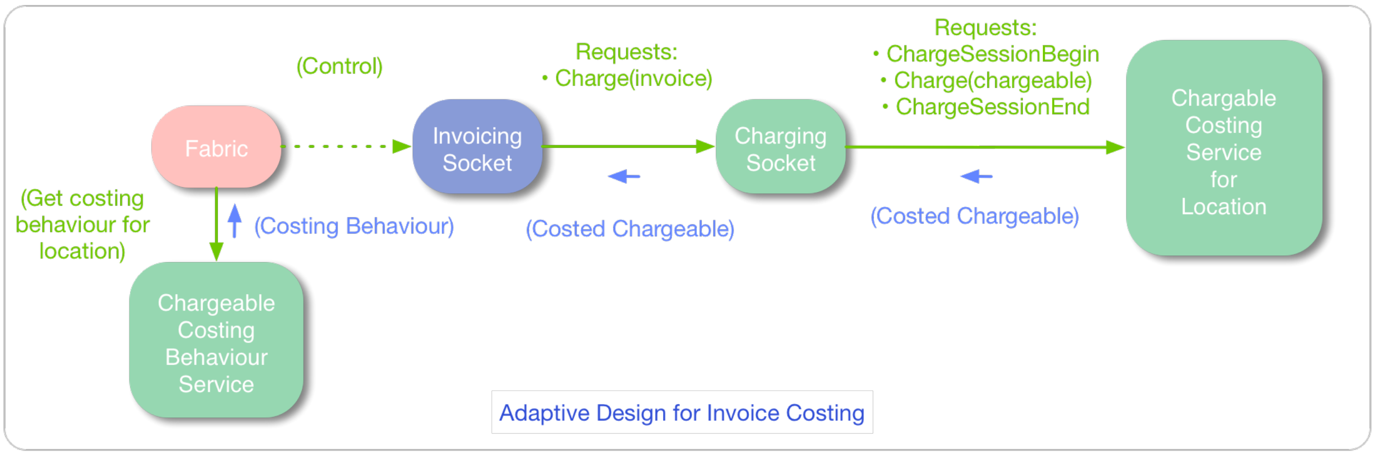
To demonstrate this injection mechanism, imagine a product cost-pricing scenario, with invoices being supplied by a **Socket** and a fabric expression which costs the chargeable(s) given in the invoice. The situational variation here is that costing varies by regional location (which conveys to variations in local tax rates, local tax laws and possibly even differences in the currency unit).

Given first is a static design, where invoices given to the **Invoicing Socket** are broken into chargeable(s), then passed to a **Charging Socket** which in turn are passed to a **Chargeable Costing Socket** which then confers with the **Product Costing Service** and a **Local Rule Service** in order to arrive at a location based product cost.



Static Design for Invoice Costing

Lets modify that design to get the Fabric to inject the specific costing behaviour, crafted and optimised for the locational variations to the **Chargeable Costing Socket**. In this adapted design, the fabric obtains the specific costing behaviour before evaluation initiates and passes it to the "**Chargeable Costing Service for Location**" element for later execution.


(Control)
Fabric
Invoicing Socket
Requests: • Charge(invoice)
Charging Socket
Requests: • ChargeSessionBegin • Charge(chargeable) • ChargeSessionEnd
Chargable Costing Service for Location
(Get costing behaviour for location)
(Costing Behaviour)
Chargeable Costing Behaviour Service
(Costed Chargeable)
(Costed Chargeable)

Adaptive Design for Invoice Costing

This design reflects a more simpler, naturally flowing topology, with no ongoing overhead in continually having to reference location context in order to arrive at a location based cost.

3.Reactive Fabric Design Process Benefits

1. Benefits From The Methodology.

The **Reactive Fabric Design Process** encourages all respective parties to engage, work and collaborate in the "**Design Space**". The project concepts and vocabulary are conceived and matured in this space. The same way Relativistic physics uses the "Thought Experiment" to ponder scenarios and their effects, the **Design Space** provides the canvas on which to play out scenarios and their effects on the whole system. In the **Design Space** functional capabilities are explored, **dependencies** are nutted out and given given appropriate residence (within the design), similarities in behaviour become visible and opportunities for optimisation are made more visible. The impact of notification bandwidth on particular heavy pathways is also considered and solutions explored.

The **Design Space** takes more of the collective group energy and in doing so, reduces the energy spent on prototyping, implementation and verification (testing). Whereas in traditional design methods, implementation and testing outweigh design, in **Reactive Fabric Design** they equally balance. The time invested in Implementation and testing is reduced, and they become more of a methodical and sizeable exercise.

Of course this is a generalisation and does not extend to every single software domain. Some special purpose software projects will always require more effort in prototyping and implementation than design, especially if they are chiefly constrained by system resources (CPU, memory, graphics bandwidth etc). The point is that: the **Reactive Fabric Design Process primarily addresses behaviour complexity**. If complexity management is not a core issue, then the benefits are questionable.

The **Design Space** provides opportunity to explore dynamic designs which adapt and morph to handle situational changes. Morphing here can refer to the use of different technologies (e.g. different network technologies to supply connectivity). Morphing can also refer to failure handling and graceful degradation. The **Design Space** widens the design palette to encompass more "what ifs" and to collaborate on what ifs.

The **Design Space** also brings much more reassurance (and orientation) to the group as it allows the group to know what it is standing on. If the design isn't addressing a feature or behaviour, then the software isn't either. The implementation mirrors the design and the design mirrors the implementation.

2. Categorisation of Implementation and Design Units.

Each software development era has come up with constructs for describing their basic software unit. The developers that embrace those constructs then become accustomed to visualising and vocalising their implementations (and possibly design) in terms of those units. Some of these major software eras and their constructs are given here below:

Era	Implementation Unit	Design Unit
Opcode Programming	The instruction (binary or assembly).	
High Level Language	The statement and function.	
Object Oriented Programming	The class object with methods and members.	UML
Patterns		The Pattern
Reactive Extensions	Sources, Observers and Observables.	The Observer and Observable.
Microprocesses		The Microprocess
Reactive Fabric.	The Element and associated patterns.	The Element and associated patterns.

3. Categorisation of Implementation and Design Units (cont).

Some of these eras were implementation centric while some follow-ons were design centric. It wasn't until the OOP era that some type of implementation model was backed with a visual/ design model (and that was UML). UML was the epoch point for a visualisation model but UML had its limitations and ended up being too heavyweight, too complex and too detailed.

As the **complier** abstracted the **opcode-instruction** into the **statement/function**,

Reactive Fabric abstracts the "**Class"** into the "**Element"**.

For the developers, the units in which they visualise with are:

- The **Element** (representing the processing of the system).
- The **Pathways** between elements (representing the relationships, dependencies, etc).
- The **Data Type** of notifications following the pathways (the type of data in the system).

For the designers the terms that they work with are:

- Topological roles of elements:
  - System boundary elements as opposed to internal elements.
  - **Control** Elements as opposed to **data processing** Elements
  - The hierarchy of Elements: the sub-systems within the system.
- The flow of data through the system
- The element dependencies as traced by notification pathways

For the architects the terms that they work with are:

- Topological Complexity.
- System Adaption.
- System Response (Notification bandwidth handling).
- Testability and Simulation-capability.

4. Benefits of Model Visualisation.

One of the major benefits of **Reactive Fabric** is that it attempts to be both an implementation and design model with each mirroring the other. In doing so, it brings a level of consistency that simplifies the work load required to conceive, design and implement.

Visualisation brings in a new dimension to collaborative prospects. Collaboration not only among software-technical staff but also across graphic designers, product management, marketing and all stakeholders.

Model sharing both horizontal (across projects) and vertical (across platforms) provides a new level of cross-coordination, integration and unification.

1. Domains of Application of Reactive Fabric Technology and SDK.

The Reactive Fabric SDK has application across a broad range of platforms:

- SOC (embedded) controllers.
- App software.
- Large applications.
- Large system software.
- Server software.

Example product bases include:

- multifunction apps and apps with multi-off-network dependencies.
- IOT devices.
- medical apps and medical devices.
- in-car multifunction devices.
- robotics control, toys, RC toys.
- audio devices and audio apps.

Example software components:

Service (provides servicing for):

- Network management such as HTTP (REST) control.
- Data services (data correlation, aggregation and formatting).
- Local hardware device servicing (data gathering, monitoring).

Operator (provides controlling of):

- View models (abstract and decouple data models from drawing/graphic models).
- Hardware devices (controlling programmable devices).

Socketing (provides processing for):

- Complex network interaction.
- Protocol stack processing.
- Remote peer network interaction.
- Multi-Path/Multi-Network technology handover.

Scoped Operators (manages):

- Application management (example scope levels: handling application levels of initialisation, security, integrity checking, presentation initialisation and business logic execution)

Producers (generate):

- Test and simulation data generation.
- Timing events and timing control generation.

5. Reactive Fabric Origins, History and Comparisons

1. History of the Reactive Fabric Concept.

Reactive Fabric stands upon a platform of seminal principles in the same way that the **automobile** stands upon the **wheel.** The concept of the **wheel** was an epoch event but on its own it didn't have full application to propel humanity. The **automobile,** as a fusion of ancillary concepts brought about personal locamotion to a level far beyond what each individual technology-facet could provide.

Reactive Fabric, stands as a confluence of the software technologies of: **Reactive Extensions**, **Micro-Processes**, **Functional Programming** and the more distant work of the **Haskell Language**, **Software Patterns**, **Sequential composition**, **UML** design and the concept of event streams. It fuses them into a software design and development approach that extends far beyond their individual scopes.

With respect to these given software technologies:

• The **Haskell** language provided the abstraction where you don't merely define behaviour in the structural domain, but also define it in the temporal realm, making the executional provisioning and behaviour separate from when it is defined. So rather than writing a function to do something, we define a function which will construct a function to do that something. This supports resource allocation at the time it is required and the customisation of the target behaviour. Haskell also provided the concept of functional composition (which also goes by other names of sequential composition and fluent programming style) and functor mapping (where function behaviour is transformed through mapping functions).

• **Reactive Extensions** coming later, drew upon Haskell concepts while melding with the ideas of the event stream and the Observe/Observable Pattern to be what we know it is 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 an implementation tool and as such does not engage in the design process. It does not intrinsically share its internal process with the developer and this separates the developer from the appreciation (and engagement with) the process that the library performs.

• **Micro-Processes**: where as Reactive Extensions provides a well defined concept framework and code interface, the Micro-Process concept has been too abstract, without sufficient support to define how messaging is performed and with no patterning or classification of the different roles and arrangements of the individual micro processes. It was an innovative concept, but lacking implementation and design backing and support.

• **UML Design** was a blueprint tool, more of a representation of the implementation rather than a description of the design. It was too low level and coupled too tightly to the target (object) language. It did not cater for levels of detail and as such, quickly became unwieldy. Its target audience was limited to software specialists.

Reactive Fabric extends the these software technologies with:

a simple and extendable modellable primitive:

*The **Reactive Fabric Element***

- The primitive supports a **Pattern** scheme to describe the various roles of **Elements.**
- It provides a **Visualisation** methodology to design in solo or through collaboration.
- Encourages dissemination of models at various **Levels Of Detail** to target all interested parties.
- Includes a concise **SDK** which allows developers to easily construct and operate their own Elements (supported by large example library set).
- Unifies design across platforms.
- Scales from small to large applications and platforms.

1. Introduction to Code Samples

Note: the various code samples given in this section are authored in the **Swift 4.2** language.

The **Reactive Fabric SDK** organises its classes into a namespace hierarchy with the base of "**Rf**". Abstract interfaces reside under "**Rf**", for example the Source interface class is denoted by **Rf.ASource<OutItem>**, the Operator as **Rf.AOperator<InItem, OutItem>** and so forth for the various patterns.

Element behaviour is defined by how notifications are handled when received. The SDK thus defines a two primitive methods, one that handles receipt of an item notification and another that handles control notifications. Both also take a Rf.VTime parameter to denote the timestamp of the notification:

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

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

The InItem generic type denotes the data-type of the received item notification. These are performant interfaces intended to streamline the code that handles item notifications, as they are the predominant notification type.
For the purposes of brevity, there is also a convenience interface method that handles both item and control notifications together in one method:

```
func onNotify(
        notification : Rf.eNotification<InItem>,
        outputNotifier: Rf.ANotifier<OutItem>,
```

Here, the notification parameter encapsulates the item and vTime stamp, outputNotifier uses the same primitive interfaces to emit notifications and requestNotifier notifies requests for evaluation termination. Note: the code samples given later use the compact form for brevity and code clarity at the expense of performance.

To emit an item of type InItem and vTime timestamp we write:

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

To terminate evaluation (with error: error at timestamp vTime) we use:

```
requestNotifier.requestEvalEnd(error, vTime: vTime)
```

2. A Simple Operator Example: The **take** Operator.

The **take** operator is one of the simplest operators, it merely propagates a number of received item notifications through to the down-stream element. This count value given as a **count** parameter of the **take** operator method.

In procedural terms, the operator must handle item notifications: emitting using the outputNotifier (i.e. passes through down-stream) **count** item notifications and then raising an end of evaluation when the count is reached (by invoking requestNotifier.requestEvalEnd**)**. It must handle control notifications: begin evaluation (control enum: **.eBeginEval**) and end evaluation (**.eEndEval**). The element is also responsible for propagating control notifications onto the down-stream element.

Take must additionally handle a special side case where the **count** is zero. The element should terminate immediately on receiving an evaluation begin**.**

2. Simple Operator Example: The **take** Operator (Cont).

```
extension Rf.Operators
{
    public static func take<Item>(_ count: UInt) -> Rf.AOperator<Item, Item>
    {
        typealias eEvalNotify = Rf.EvalScope.eNotify

        let traceID = Rf.TraceID("take")
        let takeOperator : Rf.AOperator<Item, Item> = Rf.Operators.onNotify(traceID, { (inputNotification, handler) in

            switch inputNotification
            {
                case .eEvalControl(.eEvalBegin(let evalType), (let vTime)):  // On a begin evaluation Control notification.

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

                    if count == 0
                    {
                        // Immediately request evaluation end.
                        handler.requestNotifier.requestEvalEnd(vTime: vTime)
                    }

                case .eItem(let (index, item, vTime)):            // On an Item notification.

                    let count1 = Int(count - 1)

                    switch index
                    {
                        // Propagate the item.
                        case 0..<count1:    handler.outputNotifier.notify(item: item, vTime: vTime)

                        // Propagate the item and then request evaluation end.
                        case count1:        handler.outputNotifier.notify(item: item, vTime: vTime)
                                            handler.requestNotifier.requestEvalEnd(tag: traceID.description, vTime: vTime)

                        // End of evaluation with an error.
                        default:            RfSDK.assertionFailure("\(traceID.description): Unexpected index.")
                                            handler.requestNotifier.requestEvalEnd(Rf.eError("take: Unexpected index."), vTime: vTime)
                    }

                case .eEvalControl(let (evalEvent, vTime)):     // On any other control notification.

                    // Propagate the event.
                    handler.outputNotifier.notify(control: evalEvent, vTime: vTime)

                case .eFabricTool:                    // On a tool availability event.

                    // No tools required.
                    break
            }
        })
```

The above static take function defines the behaviour of the take operator. A small amount of additional boiler plate code is required to express it as an Element instance method (in the Rf.Elements.AElement class) and to enable it to be composable with other Elements. Here is the take boilerplate code extending the Rf.Elements.AElement class:

```
extension Rf.Elements.AElement
{
    public func take(_ count: UInt) -> Rf.AOperator<OutItem, OutItem>
    {
        // Instantiate take operator.
        let takeOperator : Rf.AOperator<OutItem, OutItem> = Rf.Operators.take(count)

        // Compose Take operator with the self operator.
        compose(withElement: takeOperator)

        // Return the take operator for composure with subsequent operators.
        return takeOperator
    }
}
```

2. Temporal Operator Example: The **throttle** Operator.

The throttle operator is defined 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 actually disseminated by the fabric before evaluation and passed through to the operator in the .eFabricTool control notification.

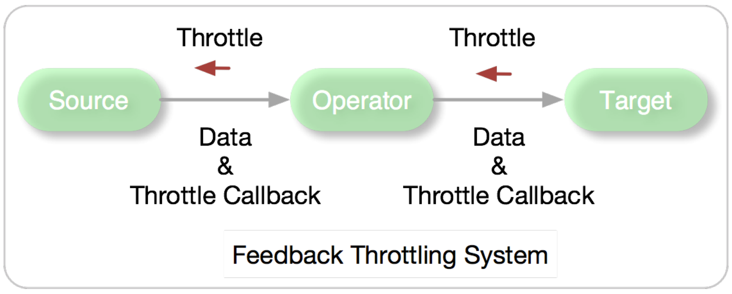
```
public static func throttle<Item>(_ duration : TimeInterval, settings : Rf.SchedulerSettings) -> Rf.AOperator<Item, Item>
{
    typealias eEvalNotify = Rf.EvalScope.eNotify

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

    // Use the onNotify operator to service input notifications.
    let throttleOperator : Rf.AOperator<Item, Item> = Rf.Operators.onNotify(traceID, { (inputNotification, handler) in

        switch inputNotification
        {
            case .eFabricTool(let (name, tools)):

                if name == "scheduler", let tool = tools.getScheduleTool(traceID)
                {
                    scheduleTool = tool
                }

            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
                }

                allowItemsToPass = true

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

                // Propagate the Eval Begin event.
                handler.outputNotifier.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
                        })
                    }

                    handler.outputNotifier.notify(item: item, vTime: vTime)
                }

            case .eEvalControl(let (evalEvent, vTime)):      // On any other control notification.

                if let scheduleTool = scheduleTool
                {
                    // Unsubscribe from the scheduleTool.
                    scheduleTool.unsubscribe(vTime: vTime)
                }

                // Propagate the event.
                handler.outputNotifier.notify(control: evalEvent, vTime: vTime)
        }
    })

    return throttleOperator
}
```

1. Notification Bandwidth Management: Scenarios Of Overactive Sources.

Notifications can be very generally classified into the use categories of **Control** or **Data**. Here **Control** notifications tend to be high level, low bandwidth directives while pure **Data** notifications tend to be primitive item-data information with a higher bandwidth level than **Control** varieties.

Data notifications can become problematic when emitting notifications that exceed the systems thread processing capacity. First-reaction measures may entail throw-away or buffering techniques which may not be appropriate. There are a variety of alternate mechanisms for handling these scenarios:

- **Employing a Feedback System between Source and Target Elements**: This entails engineering the notification pathway to be bi-directional with reply-callbacks between Elements that provide a throttle directive leading back to the source. More simply, the target Element back replies with a throttle directive to the Source and the Source then determines the best strategy. In the case where the Source is not local (i.e. remote through a network connection), then bidirectional back pathway must traverse the network interface back to the remote source. This is exemplified in the following design.

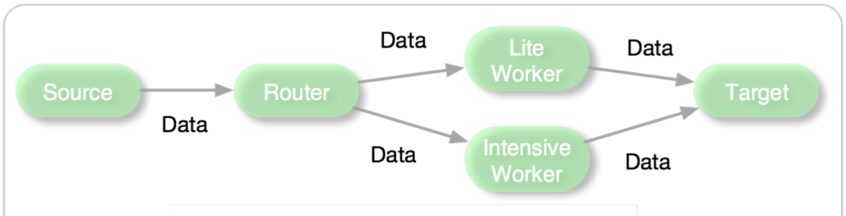


Feedback Throttling System

- **Pre-process and batch individual notifications:** at the source, batch items into a larger collective payload and notify less often with a larger payload. This works well for Audio data sources, the notification payload becomes a packet of audio samples.

- **Use Data Notifiers to supply a direct Data pathway between source and target rather than having the data notifications traverse through a series of operators**: Change the source notifications to become more high level directives rather than low-level data and engineer those directives to supply a direct data stream pathway (such as a **Rf.Notifier)**. When the target Element receives the directive with the embedded data stream, the target then engages the stream and consumes the data. This scheme works well for streaming small amounts of data such as GPS (position), mouse (position), monitoring etc.

- **Dynamically dispatch notifications along separate processing pathways**: This is an on demand **Routing** mechanism. The head element switches notifications through separate processing pathways each with its own dedicated **Rf.Eval.Queue** (each with a dedicated thread). Additionally, alternate processing pathways may also use specialised resources such as GPU processing to increase processing bandwidth.



Routing Traffic Through Specialised Pathways

2. Data Switching and Routing Strategies.

Reactive Fabric supports re-routing of data to handle scenarios such as:

- General switching of data for data capture, logging, analysis and monitoring.
- Switching inputs between real live data, test data, simulation data and generated data.
- Switching outputs between real live outputs, capture stores (in-order to capture realtime data for simulation or auditing) and output test/analysis pathways.

In order to safely perform data switching, coordination is required between the various switching points to ensure no data is lost. One method of coordination is to break the data up into data-sessions (which occur sequentially over time). In these notification pathways, rather that just messaging data items, they enforce a simple session-based protocol. This protocol employs directives of: **begin session**, **session-data transfer** and **end session**.

These directives can be represented as an item enum:

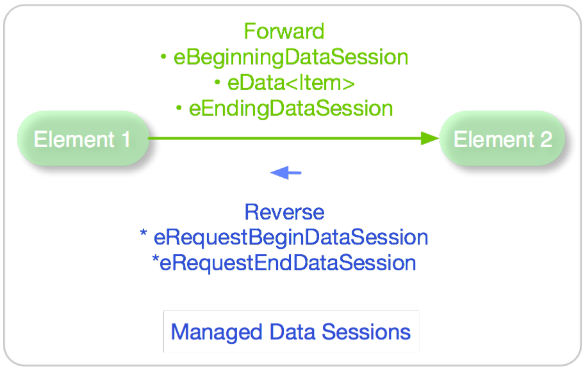
• **eBeginDataSession**	(a data session is about to begin and so data items will follow)
• **eData(Item)**	(a proper data notification within the data session, of data type: **Item**)
• **eEndDataSession**	(the data session has ended and another data session may follow)

the enum for the corresponding request replies are:

• **eRequestBeginDataSession**	(upstream request to begin a new data session)
• **eRequestEndDataSession**	(upstream request to end the current data session)

The design model for this is:

Operation: Element 2 (the downstream element) can request the start or end of a data session sourcing from Element 1. Element 1 complies by sequencing its notifications so as to only emit data notifications during an active data session. Element 1 may either back request its own sources to start and end as well or it can buffer the incoming notifications.

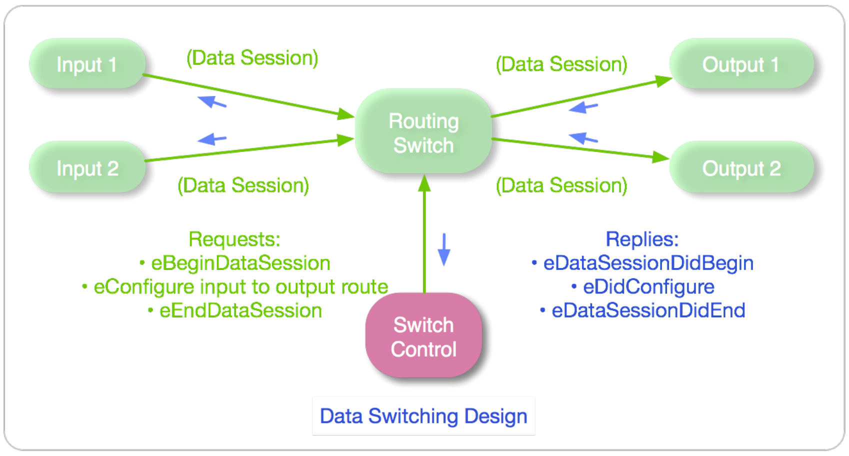


Managed Data Sessions

The full data switching design now becomes:

Here the **Switch Control** element controls the **Routing Switch** element to determine what input notification pathways (Input1, Input2) are routed to which outputs (Output1, Output2)

More specifically, the **Switch Control** issues commands to begin and end the **Routing Switch** data sessions. During **Routing Switch** non-activity the **Switch Control** commands the reconfiguration of the internal pathways of the **Routing Switch** (and then re-enables the session).



Data Switching Design