QEO SYSTEM DESCRIPTION

Qeo SDK Version 0.13.4

Revised: 27 March 2013. Printed: 27 March 2013.

Table of Contents

1 The Need for Qeo

Exchanging Information

Individual Consumer Electronics (CE) devices (smartphones, tablets, set-top boxes, TVs but also thermostats, home automation systems, doorbells,) are acquiring features and processing power at a tremendous rate. However, the usefulness of these devices is limited when they are used in isolation, and are hence limited to the information they themselves can gather about their environment.

A **thermostat**, for example, typically only makes decisions about room temperature based on the current time (time of day, day of week and a sensor that measures outside temperature. The user experience would be vastly improved if that same thermostat could know if anyone is in the house, who exactly is at home, what rooms they are in, which family member is approaching the house (e.g. based on GPS data)...

Acquiring Information

The information needed by some of these devices may already be available, but captured within the confines of some other device.

Some examples:

- The home gateway can deduce presence of persons based on the MAC address of the cell phones connected to its Wi-Fi access point.
- A Set-Top Box knows whether it is turned on and what channel it is tuned to.
- GPS receivers in cars and cell phones can figure out if someone is on the way home.

Tightening Integration

This insight has not gone lost on the Consumer Electronics manufacturers. Many are enabling tighter integration between the different devices they manufacture. All of these efforts have these distinguishing characteristics:

- Reliance on a cloud service to interconnect devices and share information
- Creation of vertical silos: interoperability is typically limited to devices from the same vendor.

For a consumer, this approach presents certain drawbacks:

- Reliance on an always-on internet connection impacts robustness.
- When devices in the same LAN communicate via the internet, needless latency is introduced.
- All of the user's data is transferred to an "unknown" third party in the cloud, which raises privacy issues.
- Vertical silos create a strong vendor lock in.

The Goal of Qeo

The primary goal of Qeo is to dramatically improve the user experience surrounding these devices.

It aims to enable horizontal interoperability within the consumer's *personal information sphere*. This means that devices should be able to share the information they acquired directly with other devices, regardless of manufacturer or device type.

Ideally, this information sharing should be possible

- Without no or as little as possible user configuration),
- Without relying on cloud services and always-on internet connections, and
- With respect for the user's security and privacy concerns.

2 High-level Features

Qeo Mission Statement

Based on Qeo's main goal, we have derived a set of high-level features. These can be captured in a single mission statement:

Qeo is an **open standard** that enables **IP connected** devices and services to easily **exchange information** in a **secure** and **private** fashion.

The current release of Qeo already accomplishes a large part of this, but some aspects are still under development:

- Features that are not yet present in this release are marked below as **[under development]**.
- Features that are present but not fully developed yet are marked **[partially present]**.

Open Standard

An open standard means that:

- A detailed specification document is available to everyone which allows people to write their own implementation of Qeo. **[under development]**
- An open source implementation is available for reference purposes. **[under development]**

IP Connected

IP connected means that:

- Qeo devices and services are able to set up an IP connection (TCP and UDP)
- When two devices/services want to exchange data:
  - One of them is able to reach the other one with a direct IP connection (TCP and UDP).
  - They are both able to reach the public internet. **[under development]**

Devices that are not IP connected can still exchange information over Qeo, provided another IP connected device provides the functionality to bridge whatever non-IP communication mechanism exists towards Qeo.

Exchanging Information

Exchanging information means that:

- Devices and services are able to provide available information (captured through sensors, computed values, internal state...) to other devices and services.
- Devices and services are able to express the type of information they are interested in. Thus, they will only read and use the information they need or are interested in.

The type and format of the information that Qeo devices and services can exchange is not strictly regulated by Qeo. This means that Qeo integrators and Qeo service developers are able to define new types of information to exchange, and have the freedom to decide how this information should be represented.

Secure

Secure means that unauthorized devices and services:

- Are not able to request information from authorized Qeo devices and services.
- Are not able to provide information to authorized Qeo devices and services.
- Are not able to snoop the information that is being exchanged between authorized Qeo devices and services. **[partially present]**

Private

Private means:

- That information exchanged between authorized Qeo devices and services remains on devices under control of the Qeo end users
- When information needs to be passed to/through an information processor (i.e. Qeo subsystems assisting in the storage or transmission of data), the data is encrypted and, as such, unusable for that information processor.

Easy

The **Easy** here has multiple meanings:

- Qeo is easy to use from the perspective of a Qeo end user. It requires very little setup.
- Qeo is easy to integrate in a broad range of existing CE products. Qeo runs on Android phones and tablets. It will also work on Linux based gateways, Linux and Android-based set-top boxes and iPhones/iPads. **[partially present]**
- Qeo is easy to use from the perspective of a developer. Developers get an API that is familiar and specific to the environment they are working in.

3 Exploring Qeo Concepts

Qeo is two things at once:

- A cross-platform, multi-language communication middleware based on the data-centric publish/subscribe paradigm.
- A set of standardized abstractions that formally describe how many real-world concepts are represented.

The communication middleware can be considered the transport mechanism, ensuring interoperability at the transport level. The standardized abstractions make sure all Qeo-enabled devices and applications describe the same concepts (user, device, temperature, video recordings,) in the same way, ensuring interoperability at the semantic level.

Without the standardized abstractions, Qeo would still be a powerful framework for the creation of one-off distributed applications. The standardized abstractions add a dimension of openness and vendor/device neutrality that elevates Qeo from a communication library to a true ecosystem.

In This Chapter

This chapter covers the following Qeo concepts:

3.1 Data-Centric Publish/Subscribe

Problems with a Distributed Environment

As stated before, Qeo's goal is to enable the sharing of information between devices. There are, however, a number of factors complicating this:

- Initially, which entities will be interested in the provided information is unknown.
- The composition of the network is highly variable: portable devices may come and go, even static devices may be powered on or off at the whim of the user.
- The network may be unreliable and slow (wireless networks with bad reception, devices going to sleep...).

The Publish/Subscribe Paradigm

The publish/subscribe paradigm, adopted by Qeo, can alleviate a lot of these problems. It introduces the notion of a Topic as a logical channel that groups related information. Information providers (henceforth called Publishers) publish information on a Topic, while information consumers (henceforth called Subscribers) subscribe to a Topic. By subscribing to a Topic, Subscribers will receive all the information that is published on the Topic.

This setup conceptually decouples Publishers and Subscribers in

- ***Space:*** Publishers and Subscribers needn't be aware of each other's location or addressing information. They needn't even be aware of each other's existence.
- ***Time:*** there is no synchronization between Publishers and Subscribers. Publishers write to the Topic whenever they have some information to share, Subscribers consume information from the Topic at their leisure. Depending on the behavioural configuration of the Topic, Publishers and Subscribers needn't even be active at the same time for communication to succeed.

Data-centricity

The plain publish/subscribe paradigm does not attach any meaning to the actual content of the messages it disseminates: it is up to the subscriber to make sense of them. Qeo goes one step further: it implements the data-centric publish/subscribe pattern.

The key insight in data-centricity is that Publishers are basically exposing their state to the world, while Subscribers are trying to reconstruct this state from the messages they receive. In data-centric communication, the middleware does not distribute messages about the state, but rather it maintains a global view of the state on behalf of the information consumers.

For data-centric publish/subscribe, this translates to the following:

- Topics have an associated data type, and represent only objects of this type
- Publishers write the current state of an object to the Topic
- The middleware ensures that Subscribers get a consistent view of the objects published on the Topic

This approach is directly beneficial for the efficiency of communication, as the middleware can now interpret the meaning of messages that are sent, and (based on the behavioural configuration of the Topic) decide to drop certain messages that are not (or no longer) relevant.

There is an even more important advantage to data-centricity: it promotes the reusability of the information that is shared. Instead of sending out messages that are relevant only to a particular use case, the information providers in a data-centric world publish facts about the world, and any information consumer can interpret those facts and derive from them the information it needs.

Topics and Data Types in Qeo

In Qeo, Topics and Data Types are tightly coupled. For each Topic, a specific Data Type is designed. These data types reside in a namespace (e.g. org.qeo.wifi) and have a name (e.g. Station). The fully qualified name of the data type (e.g. org.qeo.wifi.Station) is used as the Topic identifier, thus ensuring a one-to-one correspondence between Topics and Data Types.

Instances and Keys

Data Types are defined as composite (and possibly nested) structures. A subset of the fields of a data type may be marked as **Key** fields. Analogous to relational databases, this means that the combination of the values of these fields serves to uniquely identify an object instance on the Topic. Multiple publications on the Topic that have the same key fields are considered sequential updates of the state of a single object. Data types may also be keyless, which means that all values published to the Topic are considered to be subsequent updates of the singular object instance on this Topic.

Qeo Data Model (QDM)

Qeo is not tied to a particular programming language. In order to facilitate the reuse of data types across implementation languages, data types can be defined in a language-independent XML description language. Such an XML data type description is called a Qeo Data Model (QDM).

Code generators translate these descriptions into language-specific data structures.

The QDM format will be the form in which the aforementioned set of standardized abstractions will be formally described.

In this release of Qeo, the code generator is not yet included. Developers will have to hand-craft the Java classes that correspond to the QDMs. Some examples of QDMs are delivered in the SDK.

3.2 Types of Information

Qeo Information Types

From a Qeo point of view, there are only two types of information: **State** and **Events**. Every Qeo Topic is associated with one of those information types, and the information type defines the **Behaviour** of the Topic, i.e. the contract to which the middleware adheres with respect to information published on the Topic.

State

About States

State Topics are used to represent the current state of (physical or logical) entities in the domain. Different kinds of entities are modelled as different data types, and hence as different Qeo Topics.

Within a single Topic, there may be multiple instances of the same kind of entity. These are distinguished by their respective values for the Key fields of that Topic.

Instance Life Cycle

Instances have a life cycle: they spring into existence the first time a publisher publishes a value for this instance (i.e. this specific combination of Key field values), subsequent publications for the instance update the globally visible instance state, and finally instances disappear either because they were explicitly disposed by their publisher, or because their publisher itself disappears.

Local cache and updates

For each Subscriber, Qeo maintains a local cache reflecting the contents of the Topic. Subscribers may iterate through, or perform lookups in, this cache. The local cache is a Subscriber's interface to the Topic. Local caches are update asynchronously by Qeo, and Subscribers can choose to receive one of two kinds of update notifications:

- A simple "something changed" notification, which is useful for Subscribers that treat their local cache as a kind of lazy shared memory, where they perform lookups on an as-needed basis
- A more elaborate notification that triggers a call-back for each changed instance, providing the current instance state. This is more useful for Subscribers that implement a state machine based on the updates of the global state of one or more Topics.

Contract

State Topics abide by the following contract:

- Every instance is, at any point in time, exclusively owned by a single publisher. Over time, the actual owner of a particular instance may change, but there will never be two Publishers at the same time that can publish updates for a given instance.
- Subscribers will eventually know the latest value for each instance on the Topic. This is true even if this latest value was published before the Subscriber subscribed to the Topic.
- Subscribers will not be aware of any instances that had already disappeared before they subscribed to the Topic.
- Subscribers will not necessarily be notified of every individual update to an instance's value: when updates occur in rapid succession, a Subscriber may only see the aggregated result of the updates.
- There is no global synchronization between Subscriber's local caches: there are no guarantees that all Subscribers will have the exact same view on the Topic at a given point in time. When the Topic has been stable for a sufficiently long time (i.e. the latest publications for all instances have been propagated to all Subscribers), the contents of all local caches will eventually line up.

Event

About Events

Event Topics are used to model discrete events. Whereas State Topics model facts that hold true for a longer period of time (e.g. whether a door is open or closed), Event Topics model facts that occur at a particular point in time (e.g. someone is passing through the door). A single Event Topic models a single kind of event that can occur in the system.

Event Topics do not hold separate instances: they just represent a continuous stream of events. Hence, Event data types have no Key fields. Of course, the fields that are in the Event data type can be used to distinguish events on a more detailed level (e.g. "person X passed through door Y"), but this is an application concern: the middleware itself does not interpret event publications to that level of detail.

No local cache

Event Subscribers will not hold a local cache of events. Rather, they will be notified immediately upon arrival of an event of that event's value. It is up to the application writer to store events for later processing or retrieval if there is a need for this kind of functionality.

Contract

Event Topics abide by the following contract:

- Events published before a Subscriber subscribed to the Topic are lost to that Subscriber.
- Once it is subscribed, and the subscription has been propagated to all Publishers, a Subscriber will reliably receive all events on this Topic.
- Every Subscriber will only be notified of every event once.
- Events have partial ordering: publications coming from the same publisher will arrive in the order of publication, but there is no guarantee with respect to the ordering of events that were published by different Publishers. This means that not every Subscriber will see all events arriving in the same order.

In any distributed system, there is some communication latency. Because of this latency, it may take a small amount of time for a Subscriber to be discovered by all relevant Publishers. During this time window, publications on the Topic may be lost to the Subscriber.

The reverse is also true: when a new Publisher starts, it takes some time to discover all relevant Subscribers. If the Publisher writes some events to the Topic before all Subscribers have been discovered, these events will go lost on the as-yet-undiscovered Subscribers.

Fortunately, the time windows involved are very small (on the scale of milliseconds).

Take this into account when designing your system: it does not make sense to create a short-lived publisher (e.g. a process that starts up, publishes one event and immediately shuts down). The reliability of event distribution is only guaranteed in steady-state operation.

3.3 Interaction patterns

In this Section

State and Event Topics offer Publishers and subscribers a way to provide and consume information. In this section, we discuss the high-level interaction patterns that can be constructed from these building blocks to enable meaningful interaction between different Qeo participants.

Actors and Roles

There are two concepts that are required to describe the interaction:

- A ***Qeo Actor*** is an individual process or application that participates in Qeo interactions. In a distributed application, one could consider it to be one of the components of the distributed application.
- A ***Role*** is a well-defined set of responsibilities that can be taken up by an Actor in a distributed application. From the description of the Role, one can deduce the Topics the Actor should subscribe to or publish on.

One Actor can play multiple roles simultaneously.

Interaction Patterns

Currently, two main interaction patterns are defined:

- **Observer** Pattern, and the
- **Directed State** Pattern.

3.3.1 The Observer Pattern

Description

This is the basic publish/subscribe interaction pattern.

Roles

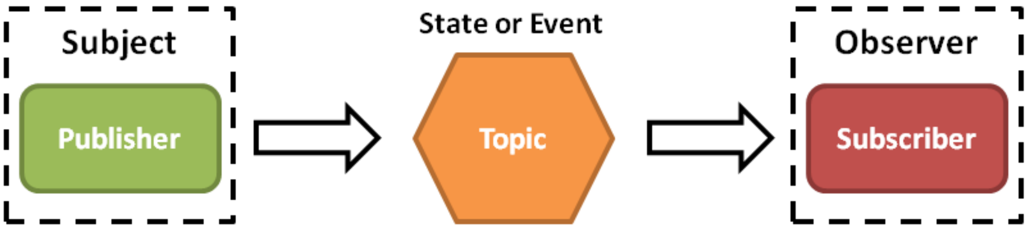
The pattern defines two roles:

- The ***Subject*** is either a source of events or an owner of state. The Subject publishes events or state updates on a Qeo Topic.
- The ***Observer*** is interested in a certain type of events or a certain kind of state. It subscribes to a Qeo Topic.

There can be many Subjects and/or many Observers on the same Qeo Topic. Subjects are unaware of the Observers, and (barring explicit modelling in the published data type) Observers are unaware of specific Subjects.

Flow of Information

The flow of information in the pattern is unidirectional: if flows only from Subject to Observer.



Reactive Programming versus Push Style Programming

This pattern fits nicely with the concept of reactive programming, where components do not push other components to perform a certain action, but rather observe their environment and react to changes in that environment. The former approach encourages close coupling of components (the push style requires the pushed component to be present or the system breaks down), whereas the latter approach is more loosely coupled, and hence more robust in a distributed environment.

3.3.2 Directed State Pattern

Description

This is an interaction pattern that allows Actors to drive the global system state forward by expressing their wishes with respect to the state of some other Actor. The key insight of the Directed State Pattern is that one Actor's wishes with respect to the global state can be considered to be part of the Actor's own state. Hence, this "desired state" can be observed, and other Actors can use this information to drive their own state forward.

Roles

The pattern defines two roles:

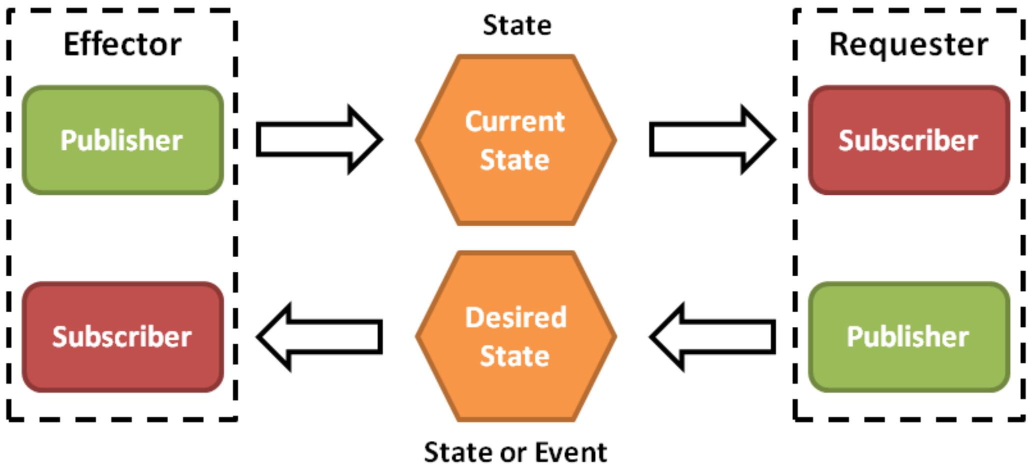
- The ***Effector*** is an actor that "owns" the state of some physical or logical entities. It publishes this state on the Current State Topic.
- The ***Requester*** is an actor that wants to influence the state owned by the Effector. It publishes its wishes with respect to the Effector's state on the Desired State Topic.

Principle

In essence, the Directed State pattern is the combination of two unidirectional Observer patterns:

- the Effector is the Subject for the Current State, but acts as Observer for the Desired State
- the Requester is the Observer of Current State, but the Subject of Desired State

The Effector will observe the Requester's desires, and based on these desires it will decide how to evolve the entities it "owns". This evolution is then reflected in the Current State, which (ideally) evolves towards the Desired State.



Advantages over the Typical RPC Pattern

On the surface, the Directed State Pattern is just a roundabout way for doing an RPC, but there are definitely differences between the two approaches:

- Instead of one bidirectional exchange, there are two unidirectional communications in this pattern. This means that it is less easy to propagate an error code back when a requested change is not possible. Rather, the Requester must observe the Effector, and conclude that the Current State does not evolve according to its wishes.
- There is no hard coupling between the Requester and the Effector: it is quite possible to formulate Desired State in such a way that it is not directed towards any specific Actor, and as such it may be possible that some other Actor A, which can also observe the Desired State, decides to drive its own state forward as a result of the publication done by the Requester. In this way, fault tolerance can be built into the system (other components pick up the work if a given Effector is non-responsive), or new behaviours may be added to the system without breaking old components (A is developed later than our Effector and Requester and implements an additional behaviour the other two don't know about).
- Because the Requester's wishes are expressed in a data-oriented way (essentially, the Requester is saying "I would like your state to evolve towards this end goal"), the purpose-specific messages of the RPC pattern are avoided. There is one generic way to express any action the Requester wishes the Effector to undertake.

3.4 The Perimeter of Qeo Interaction

IP Connectivity

The Qeo communications framework is built on top of the traditional TCP/IP stack. This limits Qeo interaction to devices that are IP-connected. At first sight, this statement excludes a lot of interesting sources of information from Qeo: home automation sensors and actuators typically communicate over non-IP networks (e.g. Zigbee or Z-Wave). These sources of information are too valuable to exclude from Qeo; so naturally, it provides a way to make them available in the Qeo realm.

Bridging

The solution to this problem is to create bridging components: components that bridge the divide between Qeo and some other ecosystem, which bi-directionally translate information and interactions between Qeo and the secondary ecosystem. For example, data exchanges within a Zigbee mesh will typically end up in some sort of hub component that makes sense of the incoming information and sends out commands to the actuators. If this hub device is IP-connected, it can translate all the incoming information into standardized Qeo data models, and publish this information onto Qeo, thus making the wealth of information available within this closed-off ecosystem available to a much wider audience.

This bridging strategy brings along the opportunity for real cross-ecosystem interoperability. For example, the Zigbee and Z-Wave ecosystems tackle overlapping problem domains, but each uses its own data model to represent the available information and interactions. By judiciously defining the standardized QDM data models for this same subject matter (e.g. temperature, lighting, etc.), Qeo bridges for each of these ecosystems can uniformly represent information and interactions. This will enable scenarios where e.g. a Z-Wave pushbutton is used to control a Zigbee light socket with Qeo as an intermediary.

The Qeo Realm

Even with the limitation of IP connectivity, we need to put stronger boundaries on Qeo interaction between devices. From both the scalability and privacy points of view, we need to limit which devices can share information with each other.

What devices exactly are contained within a user's personal information sphere? A non-exhaustive list includes:

- Consumer electronics devices in the home (gateway, set-top box, smart TV, IP-connected thermostat, etc.)
- Mobile devices, both personal (cell phones) and shared (tablets)
- General-purpose computers (PCs, laptops, perhaps a user's computer at work)
- Information stores in the cloud (e.g. a cloud-hosted calendar or the Video-on-demand service operated by the user's ISP)

Clearly, this rules out a simple perimeter like "the user's home LAN" as the boundary for useful Qeo interaction. Instead, Qeo relies on the user to define the boundaries of interaction. Users must explicitly include devices (or individual applications on general-purpose, multi-user devices like PCs) into their Qeo Realm.

A Qeo Realm can therefore be defined **as a set of devices and applications that can exchange information over Qeo**. The boundaries set by the user will be enforced through Qeo's security framework.

3.5 Privacy and Security

Goal

The goal of Qeo's privacy and security framework is to provide

- *Authentication:* reliable verification of the identity of the participants in the information exchange, at device, application and user level.
- *Access control:* enforcement of different privilege levels associated with individual users.

3.5.1 Authentication

Description

The Qeo authentication system defines which parties can participate in the information exchange within a Qeo Realm.

Identities

Qeo distinguishes three kinds of identities:

- *Device:* a physical device that is introduced into the Qeo Realm
- *User:* a user defined within the Qeo Realm
- *Application:* a Qeo Actor, installed by a User on a Device.

Certificates

The authentication system is certificate-based. This approach has several advantages over a password-based scheme:

- No reliance on a central authentication authority
- No password reset/strength/… policies needed
- Works also for headless devices (passwords don't work well on devices without keyboard and screen)
- Certificate distribution & authentication can work mostly automatically behind the scenes, which is more user-friendly.

The following certificates are issued:

- One device certificate per user on the device. These certificates are generated and distributed when the device is brought into the Qeo Realm.
- One application certificate per user per installed instance of the application. This certificate thus binds together user, application and device.

The device certificates are only used for management purposes, while the application certificates are used for the actual mutual authentication and setup of encrypted communication.

3.5.2 Access Control

Description

The Qeo access control mechanism is based on policies that restrict which Qeo Topics may be written on or subscribed to by the Realm's users. These policies are defined centrally, but distributed to the individual end points (applications on devices). The individual end points are responsible for the enforcement of the access control.

Transparent Mechanism

On the Qeo API level, the access control mechanism is transparent: the Qeo middleware layer handles the policy enforcement under the hood.

The current Qeo release only implements Authentication, not Access Control.

3.5.3 Security Management Service

Centralized Management Service

There is a need for a centralized Security Management Service that can handle the administrative task of setting up the Qeo Realm, managing users, generating and distributing certificates and crafting and distributing access policies.

This Security Management Service is hosted on the public Internet, and consists of three components:

- A user-facing front end application for configuration and management of the Qeo Realm.
- A Certificate Enrolment server that generates the certificates and distributes them to devices.
- A Device Management server that distributes the access control policies and certificate revocations.

User Interaction with the Security Management Service

The first step is, obviously, the creation of a Qeo Realm. An Administrator account is coupled to this Realm on the Security Management Service. The Administrator can then

- Add devices to the Realm
- Create users in the Realm
- Define access control policies for the Realm
- Delete users from the Realm
- Remove devices from the Realm
- Revoke individual application certificates.

The Device Management server and its associated functionalities have not yet been included in this Qeo release.

Independent Operation

Once these tasks are done, and the necessary certificates and policies are pushed down to the individual devices by the Certificate Enrolment and Device Management subsystems, no further interaction between the Security Management Service and the devices in the Qeo Realm is needed.

4 The Qeo API

About the Qeo API

Qeo applications can be written in multiple languages, and targeted to a multitude of implementation platforms. Qeo aims to provide the same level of functionality for every implementation language, but with an API and data representation that feels native for that particular language and platform.

Qeo uses a language independent API and one language-dependent API per supported language. The language-specific APIs are out of the scope of this document.

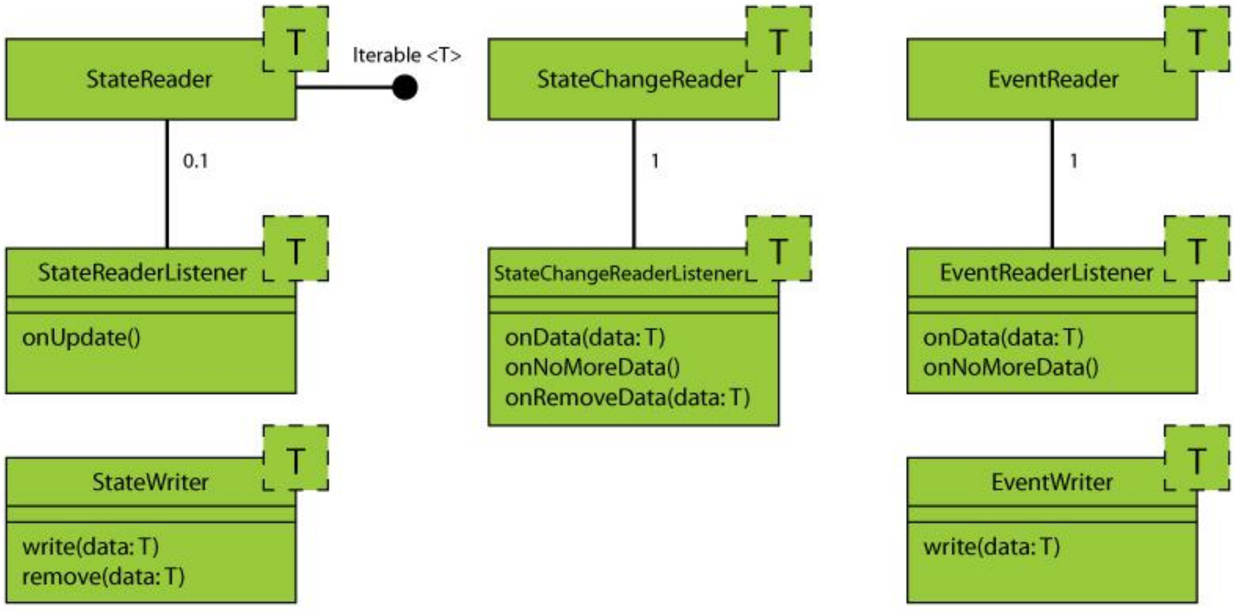
In this Chapter

This chapter covers the following topics:

4.1 Qeo API Entities

Diagram

The following diagram shows the entities defined by the Qeo API:



Entity Types

In essence, there are two types of entities:

- Writers provide data on a Topic.
- Readers consume data on a Topic. Every Reader has an associated Listener that enables the Reader to notify the upper layers of the application that it has detected some activity on the Topic.

The Writers and Readers are created for a specific Topic. This is achieved by parameterizing them with a type T, which is the Topic's Data Type.

There are specialized Writers and Readers for each Qeo Behaviour.

4.2 Writers

Overview

The following writers exist:

- StateWriter
- EventWriter

StateWriter

This entity is used to publish State information. It provides the following operations:

- **`write(data: T)`** writes a data value to the Topic. There is no distinction between creating an instance on a Topic and updating the instance with a new value. Instance creation is simply implied by the fact that no value with the same Key field values has been written to this Topic in the past.
- **`remove(data: T)`** removes an instance from the Topic. This operation only takes the Key fields of the Data Type into account; all other field values are ignored.

EventWriter

This entity is used to publish Events. It provides only one operation:

- **`write(data: T)`** writes an event value of type T to the Topic.

As Event Data Types do not have Keys, there are no instances on the Topic: all publications are considered to be one flat stream of events. Therefore, it does not make sense to have a remove operation on an EventWriter.

4.3 Readers

Overview

The following readers exist:

- StateReader
- StateChangeReader
- EventReader

StateReader

Description

This entity is used to subscribe to State Topics. It builds a local cache that holds the latest state for all instances on the Topic. The application logic can then query this cache whenever it needs information from the Topic. Currently, the only operation defined on the StateReader is iteration over all instances in the cache.

In the future, more operations will be added, including lookup of an instance based on key fields, or based on a predicate over the field values of the Topic's Data Type.

Notifications

The accompanying **StateReaderListener** offers a single notification:

- **`onUpdate()`** indicates that something has changed in the local cache (i.e. whenever the global state of the Topic has changed). The notification offers no details on the exact nature of the change; it is up to the application to iterate over the cache to find out what has changed.

StateChangeReader

Description

This entity is used to subscribe to State Topics. Contrary to the plain StateReader, it does not build up a local cache of the Topic state. Rather, it offers a notifications-only interface that is more akin to the semantics of an EventReader. The StateChangeReader is designed to allow applications to treat state transitions as events. Note that the contract of the State Behaviour still applies: there is no guarantee that the application will be notified of every single state transition. If state transitions for a given instance occur in rapid succession, Qeo may aggregate them and notify only once.

Notifications

The StateChangeReaderListener offers the following notifications:

- **`onData(data : T)`** a new value has been published on the Topic. This signifies either the creation of a new instance or the update of an existing instance. The published value is included as the data argument.
- **`onRemove(data : T)`** an instance was removed from the Topic. Only the key fields of the data argument have valid values; all other fields should be ignored.
- **`onNoMoreData()`** there are no immediate pending notifications for the Reader. This notification is sent after a series of onData or onRemove notifications. This allows applications to aggregate the incoming notifications and perform, for example, an update of the user interface just once instead of upon every single notification.

EventReader

Description

This entity is used to subscribe to Event Topics. The only interactions possible are through its attached EventReaderListener.

Notifications

This listener offers the following notifications:

- **`onData(data : T)`** a new event has been published on the Topic. The event value is included as the data argument. This is the application's only chance to register the event: after this notification, the EventReader forgets about the event.
- **`onNoMoreData()`** there are no immediate pending notifications for the Reader. This notification is sent after a series of onData notifications. This allows applications to aggregate the incoming notifications and perform, for example, an update of the user interface just once instead of upon every single notification.

END OF DOCUMENT

Appendix A Qeo Glossary

Overview

Below is an overview of the concepts in this glossary:

- Actor
- Behaviour
- Data Type
- DDS
- Directed State Pattern
- Key
- Observer Pattern
- Publisher
- QDM
- Qeo
- Reader
- Role
- Sample
- Subscriber
- Topic
- Writer

Actor

A Qeo Actor is an individual process or application that participates in Qeo interactions. In a distributed application, one could consider it to be one component of the distributed application.

Actors play one or more Roles.

Behaviour

Behaviour is a contract that describes the semantics of a given Data Type. This contract describes whether published values are disseminated reliably, whether a subscriber is guaranteed to see all published values, etc. A Qeo Behaviour maps to a set of DDS QoS settings. Currently, only two Behaviours have been defined: **Event** and **State**.

Event

Data Types with Event behaviour abide by the following contract:

- The data type is keyless
- Data values published before a Reader subscribed to the type are lost to that Reader
- Once a Reader is subscribed, it will reliably receive all values that are published for the type

State

Data Types with State behaviour abide by the following contract:

- The data type may have a (set of) key field(s) to identify different instances on the same topic
- Subscribed Readers will eventually know the latest value for each instance of the Data Type. This is true even if this latest value was published before the Reader subscribed.
- Subscribed Readers will not necessarily be notified of each individual update to an instance's value: when updates occur in rapid succession, a Reader may only see the aggregated result.

Data Type

A Data Type is the basic unit of communication in Qeo. It is uniquely identified by its name (uniqueness is guaranteed through namespacing). A Data Type definition provides information on the type's syntax and semantics.

Data type syntax is formally defined in an XML format standardized by OMG. In essence, a data type is a composite type of arbitrary complexity, in which individual fields may carry annotations. These annotations aid in the interpretation of values of the type: they indicate which fields uniquely identify instances of the type, which fields are optional, what the default values are, etc.

In addition to the syntax definition, a Data Type is associated with a well-known Behaviour (State, Event...), and potentially with some security annotations (yet to be defined). These define the semantics of the type.

DDS

The Data Distribution Service (DDS) is a powerful standard for distributed publish/subscribe communications. It is standardized by the Object Management Group (OMG). Qeo is implemented as an abstraction layer on top of DDS.

Directed State Pattern

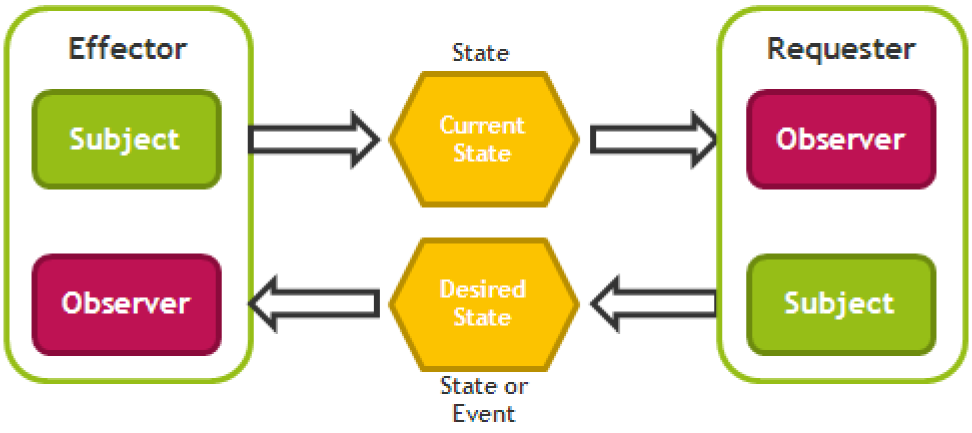
This is an interaction pattern that allows actors to drive the global system state further by expressing their wishes with respect to the state of some other actor.

The pattern defines two roles:

- The ***Effector*** is an actor that "owns" the state of some physical or logical entities. It publishes this state on the ***Current State*** topic.
- The ***Requester*** is an actor that wants to influence the state owned by the Effecter. It publishes its wishes with respect to the Effector's state on the ***Desired State*** topic.

In essence, the Directed State pattern is the combination of two unidirectional Observer patterns:

- The Effector is the Subject for the Current State, but acts as Observer for the Desired State. The Requester is the Observer of Current State, but the Subject of Desired State
- The Effector will observe the Requester's desires, and based on these desires it will decide how to evolve the entities it "owns". This evolution is then reflected in the Current State, which (ideally) evolves towards the Desired State.



Key

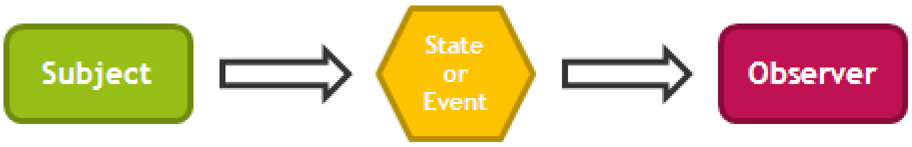
A key is a field or set of fields that uniquely identify an instance of a Data Type. Different data values with the same key represent successive values of a given instance, while different values with a different key represent different instances. If a Data Type does not have any key fields, all values are considered to be successive values of the same instance.

Observer Pattern

This is the basic publish/subscribe interaction pattern. The pattern defines two roles:

- The ***Subject*** is either a source of events or an owner of state. The Subject ***publishes*** events or state updates on a Qeo Topic.
- The ***Observer*** is interested in a certain type of events or a certain kind of state. It ***subscribes*** to a Qeo Topic.

Note that there can be many Subjects and/or many Observers on the same Qeo Topic. Subjects are unaware of the Observers, and (barring explicit modelling in the published data type) Observers are unaware of specific Subjects. The flow of information in the pattern is unidirectional: if flows only from Subject to Observer.



Publisher

A synonym for Writer.

QDM

A Qeo Data Model (QDM) is a (mostly) formal definition of the canonical way to describe a certain subject matter (e.g. Temperature, Media, Communication Sessions, Wi-Fi Diagnostics...) on Qeo. It consists of a formal description (in XML format) of a set of Data Types related to the QDM subject matter. This description extends beyond the syntax of the Data Types, and incorporates the semantics as well.

The concrete syntax for the description of the Data Type semantics is still under construction.

In addition to the formal (i.e. computer-readable) part of the description, there should also be a more informal, human-readable documentation part that describes the meaning of the Data Type fields, and the relation and interactions between the different Data Types described in the QDM.

For example, the QDM for temperature might contain a Type for temperature sensors, but also Types that describe a thermostat (a state and a request type in typical directed state fashion). The human-readable part of the QDM will then describe the typical roles in the system (thermostat-effector and requester), and how publications of request values should lead to changes of the thermostat's state value.

Qeo

Qeo is a cross-platform framework for distributed communication. It is based on the data-centric publish/subscribe paradigm.

Reader

A Reader is an entity that subscribes to a specific Topic. A Reader is an abstract concept in Qeo: in practice there will be a concrete Reader class for each kind of defined Behaviour (e.g. StateReader, EventReader...). The API of these individual Reader classes will reflect the different semantics and possibilities of the corresponding behaviours.

Role

A Role is a well-defined set of responsibilities that can be taken up by an Actor in a distributed application. For example, in a Directed State pattern, one Actor will play the role of Effector, while the other will play the Role of Requester.

An Actor can play multiple Roles at the same time (e.g. in the Video Chat example, an Android calling application will most likely implement all three of the roles, whereas a STB+webcam combo might implement only the Media Endpoint role).

For each Role, one can specify an exact list of Topics to which an Actor must be subscribed, and an exact list of Topics onto which it should publish.

At this point, Actors and Roles are defined only to introduce some convenient nomenclature for describing distributed applications in the documentation.

However, it seems that Roles in particular may be useful in a more formal sense as well: they may be described in the QDM, and some properties (perhaps security properties) may be attached to them. This is to be investigated further.

Sample

A Sample is the unit of publication in Qeo. For Event types, each new event value is a sample. For State types, every incoming state value (whether it is a new instance or a new value for an existing instance) is a sample. Disposal notifications are samples as well, although they hold no useful value.

Subscriber

A synonym for Reader.

Topic

A Topic is a synonym for a Data Type. It is more convenient to use the word Topic in contexts such as subscribing to a Topic.

A Qeo Topic is distinct in nature from a DDS Topic!

In DDS, a Topic is a combination of a unique name, a data type and a set of QoS settings (a behaviour in Qeo terms). Hence, a single data type may be reused for multiple topics, with different behaviour associated each time. In Qeo this is not the case: the topic name and the behaviour are linked hard to the data type, and hence there is a one-on-one correspondence between Qeo Topics and Qeo Data Types.

Writer

A Writer is an entity that publishes data of a specific Data Type. A Writer is an abstract concept in Qeo: in practice there will be a concrete Writer class for each kind of defined Behaviour (e.g. StateWriter, EventWriter,).