Increasing NLSR independence from sync implementations

Nick Gordon

May 3, 2017

1 Motivation

On May 3, 2017, a patch was merged that removes NSync from NLSR. In the future, NLSR will directly interface with ChronoSync using the API that it provides. During this change, a change in the internals of ChronoSync resulted in a slight change in the semantic meaning of `publishData()`. Consequently, NLSR unit tests had to be modified. Principally, unit tests should never have to change unless the fundamental behavior of the unit under test changes. However, this third-party change affected NLSR enough to force refactoring. This is clearly unacceptable long-term, due to concerns of maintainability and code cleanliness; if we cannot exercise over the shape of our own code, we cannot hope to design good systems.

It is clear that the current schema of testing in NLSR is not strictly a unit/integrated dichotomy; many, if not most, of the unit tests involve far too many different systems to be considered unit tests. However, that is not a good justification to not strive in that direction. Perfection is a habit, and we won't get to good unit tests by longingly looking at the horizon of orthogonal, well-defined components.

It is generally held that testable systems have at least two qualities: statelessness and orthogonality. We all know well enough that statelessness is really hard to do, and even harder to do when you already have lots of state. Especially difficult is statelessness in a networking system, where state is intrinsic. The other characteristic, orthogonality, is easier to achieve. One of the easiest ways to achieve orthogonality is to have interacting modules have a shared understanding, which usually means some kind of interface. Well-defined interfaces allow modules to vary independently in their implementation, provided that the still expose the same interface. However, third-party libraries rarely, if ever, will implement an interface simply because some client requests it. Then, we need some way to make a library's interface match one we expect.

This proposal offers a solution that I hope the reader finds compelling.

2 Proposal

The proposal, though not complicated, can have far-reaching effects in NLSR's codebase. As mentioned above, we cannot force anyone to adhere to our interfaces. But that does not mean we cannot adhere to them ourselves. Since NLSR's design already is heavily predicated off OOP principles, we'll refer to the OOP Bible, Design Patterns, for help. A flip through the book suggests an obvious candidate: adapters. As a refresher, adapters allows a client class to utilize some library by adapting the API the library provides to what the client is expecting.

Currently, NLSR does not expect any kind of interface to a sync protocol; interaction is done directly with the sync protocol class, and both pieces must be considered before changing. Part of this proposal is to refactor NLSR so that all work to be done by a sync protocol will be directed through a common interface, such that any arbitrary program that implements this interface with the same semantics can be substituted at will. This could be accomplished with a sync protocol by including the library and writing an adapter that implements this interface. The adapter class then would be used by NLSR in some fashion to provide sync services to NLSR and its modules.

The implementation details could vary: a type-driven system that employs CRTP (curiously-recurring template pattern) would provide a mix-in/interface that any module that needs sync services could inherit from. Alternatively, a more orthodox approach of a hierarchical inheritance tree could provide objects that provide the service to NLSR, with NLSR acting as the manager of the resource, much as it does now for other objects.

The type-driven, CRTP approach would involve creating an templated SyncAdapter class that uses static_casting and templating to access the subclass's functions. Then any class that needs to access the sync service inherits the Adapter class, providing the necessary interface in-class. Further, the Adapter could be written as a Singleton, ensuring that all inheritors access the same, configured sync instance.

The alternate approach is to define some class, say SyncAdapter, with all pure virtual functions, that adheres to the interface specified during design. Then, NLSR can simply manage the object and provide a getter method, for example.

3 Anticipated Results

It is anticipated that this change would increase the independence of NLSR and its submodules from the specific implementation of a sync protocol. We see this as the case because all modules use the same interface to the sync system, ensuring that changes in some sync implementation do *not* require changes in internal NLSR code. Instead, the changes would be made in the Adapter, such that the Adapter continues to provide the same semantic and syntactic interface to NLSR. This also increases testability of the system, as the implementation and abstraction of the sync protocol are decoupled. In particular, a fake Adapter implementation could be written that allows units that are using sync services to be tested only on their black-box interaction those services. We can do this because we are *also* testing the specific implementations for compliance with the corresponding sync protocol. Finally, because the two units are known to work, and their interface is well-defined, it can be concluded that they will work in tandem.

This provides a system through which new, experimental sync protocols can be made to work with NLSR by writing minimal code to adapt their interfaces to the one we have defined for NLSR to use. As such, it would be possible to write automated tests to change the choice of sync and gather performance metrics as well. True, it may be more difficult in some cases to write the Adapter code, but this cost must invariably be cheaper than the cost of refactoring every involved NLSR module to understand the new sync protocol.

With regards to the specific implementations offered, the advantages of the type-driven approach are that it uses static polymorphism, meaning better performance, and it improves data separation. Modules that need sync services can inherit the mix-in, and the supplied private functions are available only to the inheriting class. The most significant disadvantage of this is the amount of refactoring needed to accomplish this, because NLSR currently follows the second solution's paradigm.

The member attribute and inheritance-driven approach's advantages are the lesser effort involved in the refactoring, because NLSR currently makes extensive use of the member attribute approach, as well as having a slight advantage of complexity over the type-driven approach, in that the Adapter superclass does not need mechanisms to ensure uniqueness. The disadvantage of the approach are that is has much worse data separation; to access sync services, you must necessarily have access to all of the other objects that NLSR offers. This provides the opportunity for modules to violate their "theoretical" interface because they have access to so many other components of the system. Additionally, as mentioned above, runtime polymorphism of this sort is slower and unnecessary, because the choice of sync protocol will be known at compile-time.

4 Conclusion

It should be clear now that, from an engineering perspective, refactoring NLSR to provide sync "as a service" through a well-defined, decoupled interface will have strong positive effects on NLSR's maintainability and code quality. Though not a very attractive proposal for research understanding, it is nevertheless important; if NLSR is to become production-quality code used to route through large, complicated networks, we should strive to make the code maintainable, understandable, flexible, and fast. This solution has the potential to provide improvements in every one of these characteristics.