Telephony Data Handling Pattern Language

David E. DeLano
AG Communication Systems
delanod@agcs.com
2500 W. Utopia Rd.
Phoenix, AZ 85027
(602) 581-4679

Abstract

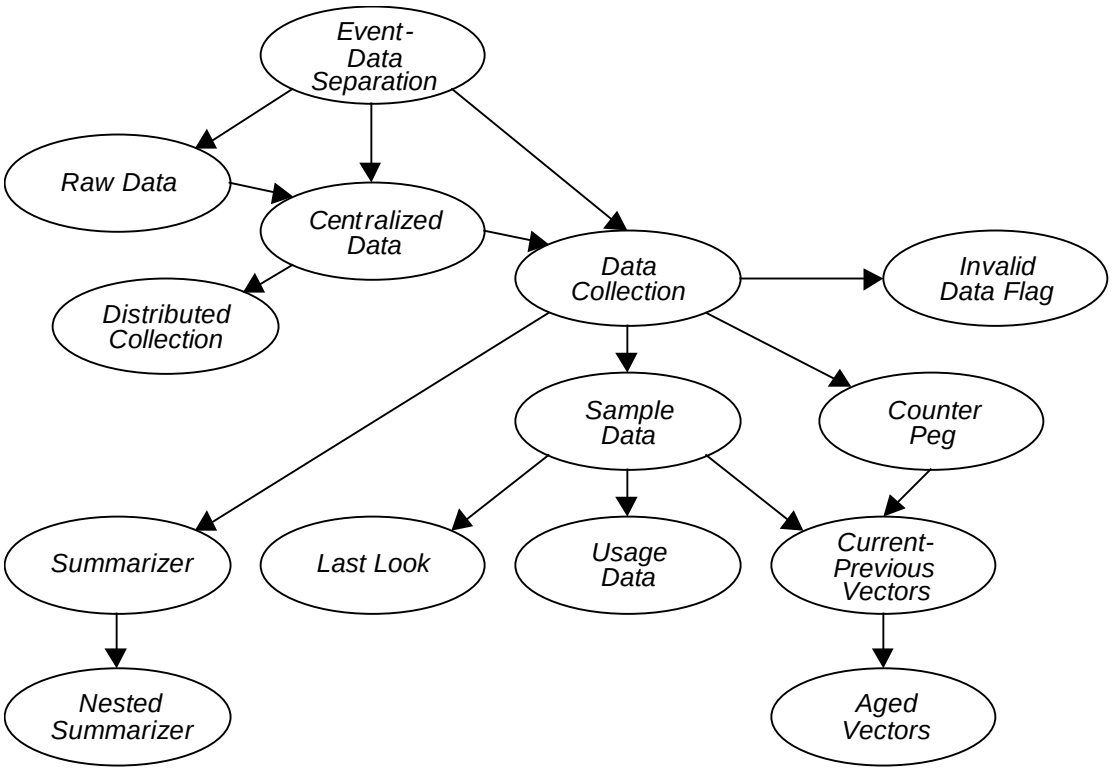
This paper is the beginning of a pattern language within a pattern language. It is intended to be used in the context of a more encompassing telecommunications pattern language. On its own, it is a pattern language of data handling patterns. These patterns may not be unique to the data handling needs of telephony systems, but are not validated outside this context.

Introduction

Telecommunications is a wide area of interest within the Patterns Community. The development of telecommunications systems has been going on for over twenty years, and patterns are beginning to appear on the design and use of such systems. A Telecommunication Pattern Language would be huge, and is best addressed as a set of related and interwoven pattern languages. It is in the context of this greater Telecommunication Pattern Language that the Telephony Data Handling Pattern Language is intended to be considered. While there may be some use of these patterns in other domains, no attempt has been made to validate the patterns outside of the telecommunications domain.

In the world of telecommunications, two things are constantly occurring in parallel: the events that are the main purpose of the telecommunication system, and data that is being collected about those events. The events are usually real time intensive and real time critical. The data collection is not real time sensitive, and should never get in the way of processing the events critical to the very nature of the system. The events take up memory only for the moment. Data collection often uses huge amounts of data that are stored for long periods of time. It is this dichotomy of event processing versus data processing that leads us into this Pattern Language for Telephony Data Handling. The language starts off with *Event-Data Separation*, which leads us down the path of resolving the event versus data conflict.

The Pattern Language



The patterns in this collection in order of their appearance are:

Event-Data Separation – How to separate event processing from data collection
Raw Data – What data should be stored
Centralized Data – Where data should be stored
Distributed Collection – How to gather collections of data for reporting
Data Collection – How data should be collected
Counter Peg – How event data should be updated
Sample Data – How periodic data should be updated
Usage Data – How busy device data should be updated
Last Look – How to handle constantly incrementing counters
Invalid Data Flag – How to protect data integrity
Current-Previous Vectors – How to handle simultaneous collection and reporting of data
Aged Vectors – How to handle the storage of historical data
Summarizer – How continuous data should be handled
Nested Summarizer – How large data samples should be stored

Known Uses

The patterns contained in this language are derived from the proven practices of a number of switching system developed at AG Communication Systems, Lucent Technologies, Siemens AG, and several smaller telecommunications companies. Those in the industry from other vendors will likely recognize one or more of the patterns in the language.

Event-Data Separation

Problem

How should a system be partitioned to handle the collection and reporting of telephony data?

Context

The primary purpose of a telephony system is to route calls. However, there is also a need to measure different aspects of the call routing, and therefore data must be collected pertaining to each call. This data is usually state or event driven, that is, a count is collected each time a state is reached or an event occurs. Sometimes the data is a sampling of data for simultaneous calls in the system. It is usually not important that this data be kept on an individual call basis, but an accumulation of call data is needed over various periods of time.

The data collected is used for three purposes. The first is a secondary, though very useful purpose. It is used for the testing and validation of the switch software. Errors can often be detected and identified by studying the call routing data. The next two are the primary reasons for collecting the data, and they are the reasons that data collection is a feature of the telephony system to begin with. First, the data can be used to identify network problems such as bad equipment or incorrect interfaces. Second, the data can be used to study traffic patterns for the projection of future system needs.

Forces

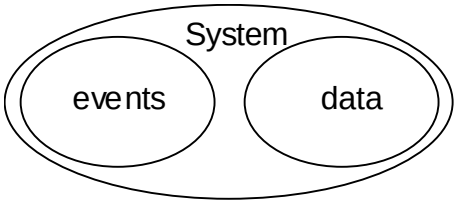
- Foremost, a telephony system is a real time system. Data collection and reporting should not interfere with the primary function of switching calls.

- The memory needed to store the collected data can be very large. It is usually not critical that this data be persistant, but some data may need to be retained for historical purposes.

Solution

Partition the data handling of the system separate from the handling of the events being measured. This should not only be accomplished by separating these functions into individual hierarchies of code or sets of objects and classes, but also as separate tasks or processes so that the real time needed to process the events can be controlled independently of the processing time needed to handle and report the data.

This system of collecting data is known as Traffic Data Administration (TDA), Performance Monitoring (PM), Network and Switch Management (NSM), and others.



Resulting Context

By starting into this pattern language, a design decision is made to separate the concerns of processing calls and call events from the collection of data associated with them. New design decisions are now presented, such as what data to collect, how to store the data, and how to bridge the border between the occurring

events and the data collection. The problem of what data to collect is initially addressed in *Raw Data*. The design for storing the data starts with *Centralized Data*. Begin design of the event-data interface with *Data Collection.*

Rationale

It was probably fairly obvious to early switch designers to partition out data collection as a separate function. Functional decomposition drives the design in this direction without much forethought. An object-oriented analysis of the problem space, though, may drive the design in the direction of including the data with the classes handling the events being measured. While a parallel universe of classes, each event handling class having a corresponding data handling class, may not be desirable design for most parts of a system, it is one of the best solutions for handling the event-data paradigm.

The event handling part of the system is the most real time critical. Though it will not always take even a majority of the available real time of the system, it is crucial that as much of the available time be given to handling events even to the extent of shutting down other parts of the system. The data collection part of the system will use up most of the memory of the system. Separating these two concerns allows the allocation of real time to the processing of events, and the handling of data storage in the most efficient manner. It should also be noted that this separation allows for changing the way the data is manipulated and stored without affecting the design of the event handling system.

Raw Data

Problem:

What data should be collected and stored in the system?

Context:

The design of the system has followed *Event-Data Separation*.

There are many requirements for data in a telecommunications system specified by the customer and by standards bodies, such as Bellcore. However, these requirements rarely specify how the data is to be stored and presented. The reports of the data are read by humans and are often sent to another system for storage and processing.

Many times there are calculations that are made with this data either by a human or as part of downstream processing by another system.

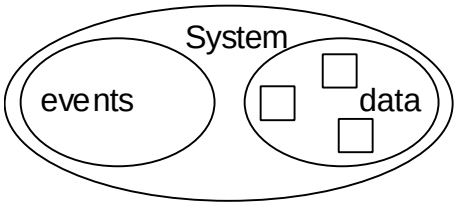
Forces:

- Often, huge amounts of space are needed. Every effort should be made to minimize to amount of memory needed to store the data.
- Calculations on the data collected take time to perform, time that is better spent processing events.
- There are often historical and cultural manipulations of the data. These should be honored even if the switch could perform this task.

Solution:

Store data in its raw form without doing any calculations. Present data with a minimum of formatting and text. This will not only minimize the time spent handling data, but will keep the data storage to a minimum. The presentation should be understandable, but no interpretation of the data should be done except as specified in the requirements.

It is tempting to present the data in some alternate format, for example, to present an average of several pieces of data or to add two data counts to present a third. This temptation should be avoided, unless such a calculation is required. This calculation may be done with the data, but for cultural and historical reasons, let the humans or downstream processing take care of this. If the temptation for doing this work is too great, at least package it into a separate report so there is an option of viewing the data.



Resulting Context:

With the problem of what data to collect resolved, there is still the problems of how to store and collect the data. To resolve these issues, consider *Centralized Data* and *Data Collection*.

The data that is stored as a result of applying this pattern is very cryptic and often useless on its own. It is up to some other process acting on the *Centralized Data* to interpret the data and provide a meaningful representation of it to the user.

Rationale:

Early telecommunication systems produced a minimum amount of data. Administrators of these systems produce calculations and table lookups for transforming this data into what they needed. As newer systems replaced the old, the expectations for receiving this raw data remained, even though modern systems are more than capable of presenting the end result of the calculations. As downstream processors were added to the network, they were designed to match this interface. Nothing more is expected from the switch, so nothing more should be presented.

Centralized Data

Problem:

How should data be stored?

Context:

Raw Data needs to be collected when specific states of a call are reached or specific events occur. This data is accumulated for multiple calls. The events and states occur in a single processor or processors with a shared memory area. Individual call data is not needed.

Forces:

- The memory needed to store the data grows very large, very quickly. Management of this data space should be simplified as much as possible.

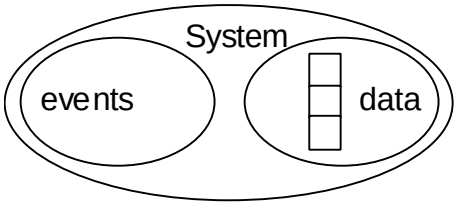
- The data needs to be reported in a cohesive manner. Data that is reported together should be kept together.

- The time to process the data for a report should be minimized. Time is valuable for event processing, not data manipulation.

Solution:

Store data that is used together in one location in some collection that shall be called a vector. This way the data can be processed or shipped to another location in one piece. The actual storage could be an array or a collection of named variables that are contiguous in memory. It is important that the data be stored in a manner such that the entire vector can be sent elsewhere for further processing or storage. All *Raw Data* that resides in a shared memory space should be stored together.

Note that the memory used for this storage should be easily and quickly accessible. Storing the data into a database or other repository is not a good solution. This is better left to *Distributed Collection.*



Resulting Context:

There must now be an efficient method for updating the data. Consider *Data Collection* for this purpose. Other considerations of storing the data must also be addressed. There are most likely several *Centralized Data* stores that must be coordinated and combined using *Distributed Collection*.

Rationale:

Data that is used together should be kept together. Storing the data in a vector format simplifies, thus using less real time, transporting, reporting and manipulating the data. This includes activities such as zeroing the data.

Distributed Collection

Problem:

How should data that is distributed across multiple locations be gathered for reporting?

Context:

Data is stored as *Centralized Data.* However, the data is kept in multiple locations because it is controlled by separate processors or processes. The data may even be kept in multiple systems.

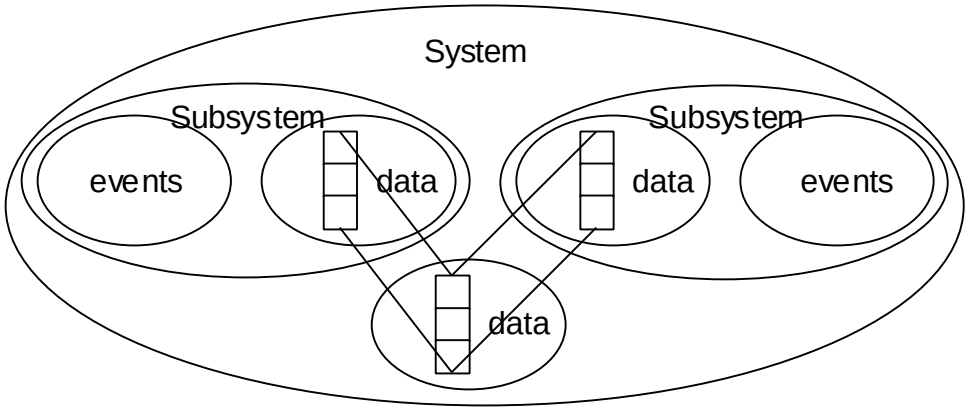
Forces:

- Reporting of data is a non-real time activity, yet it should be executed as efficiently as possible.
- Multiple reports may use a common set of data.
- Reports may be requested for historical data.
- The memory needed to store the data should be minimized.

Solution:

Create a process and accompanying data storage for accumulating the data from the various *Centralized Data* locations. This storage can reflect the *Centralized Data* structures, if the data needs to remain separate for each process or processor. If separate data is not needed, the data can be accumulated as it is collected.

On a periodic basis, send out a message to each of the *Centralized Data* processes requesting that the data be sent in for storage. This message should be handled in a non-real time manner so that it does not interfere with the accumulation of data from ongoing events.



Resulting Context:

The data from the system is now collected into one place and ready for reporting. For the most part, though, the *Distributed Collection* represents yet another copy of the data, and thus uses more memory.

Current-Previous Vectors may be needed to store the data if reporting on the data needs to coincide with collecting a new set of data. It may be possible, however, to delay the current report until the new data is collected.

Aged Vectors may be used if more than one collection of data is needed for historical purposes.

Rationale:

While it is good to keep data accessible to the event subsystem for updating, the reporting of this data becomes cumbersome if it is spread all over the place. Periodically accumulating the data into one place greatly aids the reporting process and allows for more massaging of the data if needed. The storage location for the *Distributed Collection* may even be in a processor or process completely separate from the event subsystem.

Data Collection

Problem:

How should data be collected from the event system?

Context:

Raw Data is stored as *Centralized Data*. The system is separated using *Event-Data Separation*. Events occur that must update some data.

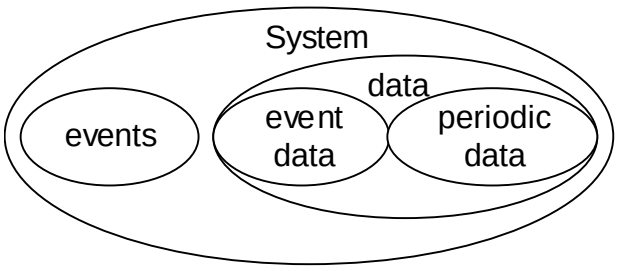
Forces:

- As little real time as possible should be expended updating data.
- Different kinds of data need to be collected, which require different data collection methods.

Solution:

Separate the data collection into two categories. One set of data will be collected on an event driven basis using *Counter Peg*. The other set of data will be collected periodically using *Sample Data*.

There may be a need to combine the two if periodic samples of event driven data are needed. In this case, *Counter Peg* is used to collect the data that is then periodically collected using *Sample Data*.



Resulting Context:

Decoupling of the events and data collection is maintained. The choice of whether to use *Counter Peg* or *Sample Data* still needs to be made based on the type of data available.

The possibility of overflowing or underflowing a counter must always be considered. Counters should be sized to minimize this possibility, but checking for these conditions is required to maintain valid data. Should such a condition occur, the counter could be frozen at its maximum value, be frozen at zero, or be allowed to wrap around to zero and start counting again. The last case is useful if *Counter Peg* is used to gather data that is periodically collected as *Sample Data.* In any case, the use of *Invalid Data Flag* should be considered to protect data integrity.

Rationale:

Data collection methods are used which match the type of data being collected. The system is not encumbered with conforming to one interface for all data collection.

Counter Peg

Problem:

How should access be handled to allow for the updating of the event driven data?

Context:

It has been determined that *Data Collection* is required. At specific points in the event handling, a counter must be updated or "pegged". This is usually done by incrementing the counter by one. Less frequently, a counter may need to be updated by more than one, in which case *Sample Data* should be considered. The data may also need to be decremented, or "unpegged" if it is recording the level of current activity or if an event is being undone, as in some error handling schemes.

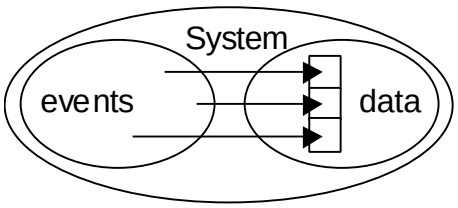
Forces:

- As little real time as possible should be expended updating data.

Solution:

Design an interface that allows easy and quick updating of counters. This can be done through macros or inline object methods, for example. No calculations should be done during this update except for counter overflow/underflow conditions. No other public access to the counters is allowed.

This solution can be visualized as the event processing subsystem "pushing" the data from the event handling side of the system to the data handling subsystem.



Resulting Context:

An efficient method for writing to a counter has now been established. If it is more desirable that the data subsystem "read" the data from the event subsystem, then *Sample Data* should be considered.

There are other conflicts with writing the data that could cause the algorithm for pegging a counter to be inefficient. These include access to the data for reporting purposes while counters are updated. *Current-Previous Vector* addresses this issue.

While the design of the data handling system has allowed the decoupling of event and data handling, a stronger coupling has now been introduced. It is important to recognize the existence of this coupling and to protect from making it worse.

Rationale:

Because the processing of events and the handling of data are separated into different time spaces there needs to be an efficient interface for updating data. This means the border between the event handling and data handling needs to be more porous than is usually desired. The porous nature of the interface should allow for minimal real time impacts, yet protects against misuse of the data.

Sample Data

Problem:

How should collection of data samples be handled?

Context:

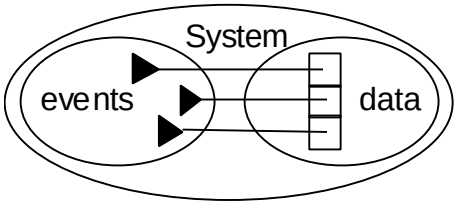
The system has counters that need to be sampled on a periodic basis. The counters may be stored as a hardware register or using *Counter Peg*.

Forces:

- Real time to collect the data is of utmost concern. The impact to data collection on the event handling subsystem should be minimized.

Solution:

Have the data collection subsystem sample the data on a periodic basis, at a lower priority than the event handling subsystem. This can be thought of as "pulling" the data from the event handling system. Any manipulations of the data can be done during this non-real time critical processing.



Resulting Context:

Now that a method for collecting the data is established, the data must be stored in an efficient manner that protects the integrity of the data. *Current-Previous Vector* should be considered for this purpose, though *Summarizer* may also be appropriate.

If the data being collected is really a counting of events instead of a measurement of active events, *Counter Peg* should be used. If the data being collected is based on a count that is incremented and decremented, giving the count of devices in use, for example, *Usage Data* needs to be used to extend *Sample Data*.

In the case of collecting data from a wrap-around counter, *Last Look* will be needed for determining the value to store.

Rationale:

Many event counters are kept in a hardware register and it is not appropriate use of real time to handle an interrupt each time the counter is incremented. Therefore, a mechanism of polling for this data is more appropriate.

Usage Data

Problem:

How should data be stored for calculating busy device data?

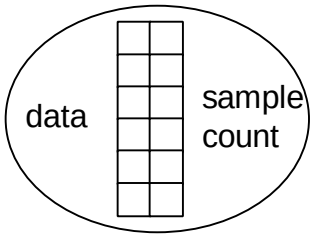
Context:

The system has counters that represent a measurement, for example the number of devices of a certain type that are in use. *Sample Data* is used to periodically collect this data, but this may be insufficient for making the calculations necessary for reporting of the data.

Forces:

Solution:

The data vector is kept along with a count of the number of samples that have been collected. When the data is processed, these two pieces of data can be used to calculate an average, or a more sophisticated derived data such as hundred call seconds.



Resulting Context:

Additional memory is needed to store the sample counts, but accurate calculations can now be done with the data. It cannot be assumed that the amount of samples taken in a fixed time interval is the same. Since this collection is done without affecting real time, samples are often delayed causing few samples in one time interval and more samples in the next.

Rationale:

There are many telephony-oriented calculations that need to be made to determine how well a system is engineered. These calculations often are made based on the amount a time a set of equipment is kept busy, or the number of devices that are busy over time in a set of similar equipment. These calculations are then used to reengineer the system to make it more efficient.

Last Look

Problem:

How do you calculate the value that needs to be stored in *Sample Data*?

Context:

Data Collection has been used to determine that *Sample Data* needs to be collected. However, the data is being collected from a hardware register that is not reset after reading it. Instead, the counter increments to its maximum value, then starts over from zero. This is often referred to as a wrap-around counter.

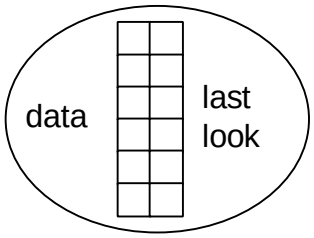
Forces:

- The collection of the data must remain unobtrusive. Collection must proceed with the data available.

Solution:

Maintain the last value read from the counter as a separate data item. The current value that needs to be collected and stored can be derived by subtracting this value from the current value read. The algorithm needs to adjust for the case when the counter wraps around. This adjustment will be slightly different depending on whether the counter wraps to zero and zero needs to be counted, or the counter wraps to one.

Software generated *Sample Data* could also have the same context and require a *Last Look*.



A example calculation could go here.

Resulting Context:

The *Sample Data* that is needed is collected without interfering with the existing event *Counter Peg* mechanism. Additional work for the event system is also avoided. All of the processing needed to collect the data occurs in the periodic, non-real time critical part of the system.

Rationale:

Sometimes you have to live with the counter mechanisms that are given to you, especially if they are hardware and cannot be changed. Instead of imposing additional requirements on the event system to make data collection easier, the data system needs be able to use the data that is available.

Invalid Data Flag

Problem:

How do you maintain the validity of the collected data?

Context:

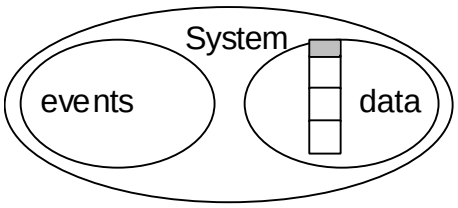
Counter Peg and *Sample Data* are storing data as *Centralized Data*. Because of various conditions, the data currently being collected may not be in a valid state.

Forces:

Reporting valid data is critical. Impartial or invalid data should still be obtainable, but must be reported as potentially being incorrect.

Solution:

Keep a flag with the data that can be set when the state of the data is potentially invalid.



Resulting Context:

The data is reportable, but is marked as a potential problem.

Rationale:

The personnel who operate a telephony system want to get data from the system regardless of the state of the data. However, it is important to provide notification that the data may not be accurate so that no drastic actions are taken as a result of misinterpreting the data.

Current-Previous Vectors

Problem:

How do you maintain the consistency of data over time, when data needs to be captured over a specific time interval?

Context:

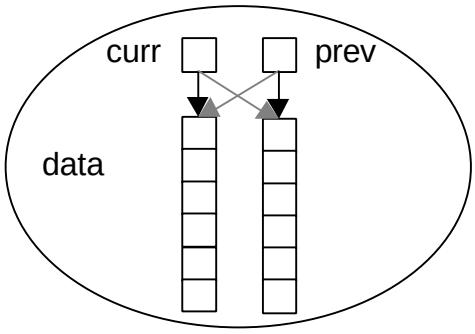
Raw Data is being stored as *Centralized Data*. The vectors in the *Centralized Data* have an *Invalid Data Flag*. *Distributed Collection* is used to gather up the various pieces of *Centralized Data* into one location on a periodic basis.

Forces:

- The storage of the data needs to be kept to a minimum. The use of any intermediate data storage should be minimized or eliminated.
- All of the data in one storage vector needs to be kept over a consistent time period.
- The real time processing of events should not be interrupted, or the interruption should be minimized.

Solution:

Use two identical vectors for storing the data. One copy, the current, is used for *Data Collection*. The other copy, the previous, is used for reporting data. The switch between these two vectors should be protected so that *Data Collection* is not occurring during the switch. The processing of the switch proceeds as follows: The previous data vector's *Invalid Data Flag* is set; The previous data vector is cleared; The current and previous pointers are switched; The current data vectors *Invalid Data Flag* is cleared (this was the previous data vector *Invalid Data Flag* before the switch). At this point, the event handling subsystem is free to continue *Data Collection* and the *Distributed Collection* continues using the data referenced by the previous data pointer.



Resulting Context:

Using *Current-Previous Vectors* will require twice the data storage of simple *Centralized Data*, but will allow the event subsystem to continue updating counters at the same time as *Distributed Collection* or data reporting. The integrity of the data is preserved.

If historical data needs to be preserved, *Aged Vectors* need to be used. Note that it is possible to have an *Aged Vector* that contains only two vectors, which behaves very much like *Current-Previous Vectors.*

Rationale:

It is so important that the event subsystem not be interrupted that schemes for minimizing the real time impact of reporting data take precedence over the amount of memory needed to store the data.

Aged Vectors

Problem:

How should consistent data be kept when more than one historical copy of the data is required?

Context:

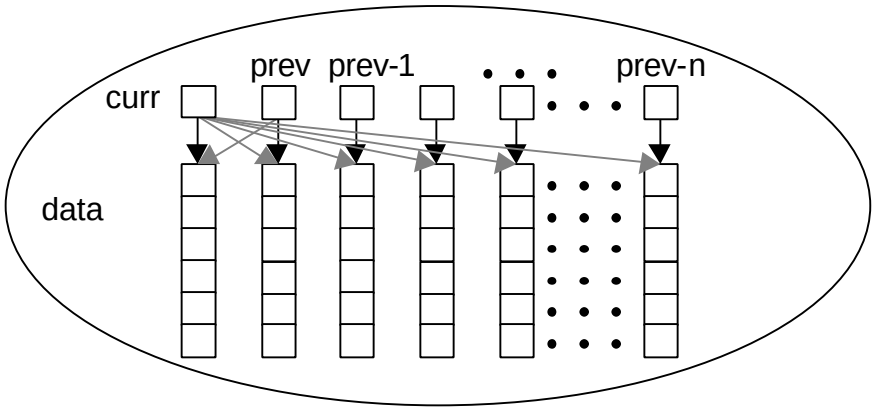
This pattern shares the same context as *Current-Previous Vectors* with the additional requirement that more than one copy of historical data needs to be kept.

Forces:

See *Current-Previous Vectors.*

Solution:

Use multiple identical vectors for storing the data. One more than the number of significant data samples should be stored so that one, the current, is being updated. The rest are used for reporting historical data. The most recent of these is referenced as previous, with the others being referenced as an offset from previous. Switching of the vectors is similar to *Current-Previous Vectors*, except that all pointers need to be rotated to the next data sample. Only the oldest data sample is cleared, in preparation to becoming the current data vector.



Resulting Context:

Depending on the quantity of data being collected and the number of samples being retained, the data storage needed will be quite large.

If individual samples of data need to be kept instead of historical sums of data, use a *Summarizer*.

Rationale:

Most telephony based systems have requirements for keeping historical data. This data should be immediately available and not be stored away in such a manner that it is difficult to retrieve. If multiple processors or processes are involved it may be possible to store the *Centralized Data* using *Current-Previous Vectors* and only use *Aged Vectors* for the *Distributed Collection*.

Summarizer

Alias: *Sliding Window*

Problem:

How do you store and represent data that is accumulated over time?

Context:

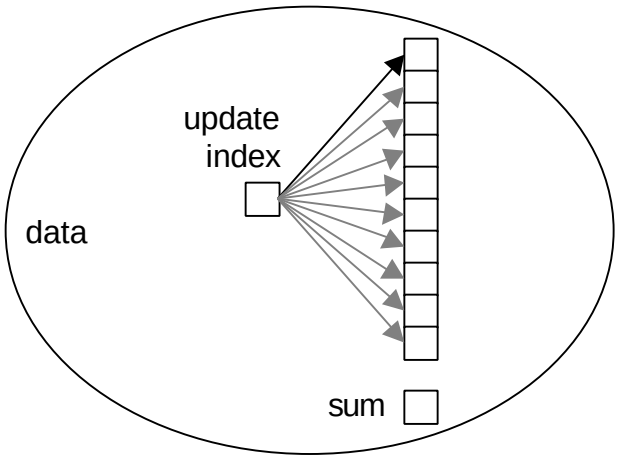
In addition to keeping data in the form of counts for events and states, it is also necessary to keep sample counts of data. These samples can't use *Counter Peg* because they aren't adding one to a count. Instead, a sample of data needs to be captured periodically with a sum or average of the collected data calculated with every sample update.

Forces:

- Data storage needs to be taken into consideration. This is especially true because multiple data storage units are needed for each type of count being sampled.
- Real time to collect the data and store it is not as critical. Samples are usually collected passively instead of when an event occurs.
- Multiple calculations may be needed on the data, so efficient calculation methods should be used.

Solution:

Store the data as a collection of samples that are stored via an *Iterator*. Once all the units in the collection are full, replace the oldest sample with the current sample. At a minimum, provide a sum of the entire collection that can be returned to the calling program or retrieved. Note that the sum can be easily calculated by subtracting the oldest sample and adding in the newest sample, instead of adding up all of the elements each time.



Resulting Context:

A *Summarizer* is good at collecting a reasonable amount of data, say ten samples, but the memory to store large amounts of data is very large. If a 100% accurate representation of the data is needed, there may be no other choice but to use a large *Summarizer*. However, if data accuracy can be sacrificed a bit, a *Nested Summarizer* should be used.

If transient faults are being monitored, it may be more appropriate to use a Leaky Bucket [PLOPD2].

Rationale:

Requirements for detecting errors over time depend on a sliding window of data that represents the errors that have occurred in a fixed window size. Old data is not needed in this calculation, so it is replaced. The sum of the data is needed to compare to a fixed threshold. When this threshold is crossed, an event is usually generated that causes some sort of recovery. Thus, a quick determination of the sum of the samples is needed. For other applications, an average of the data samples may be more appropriate.

Nested Summarizer

Problem:

How do you represent a large number of data samples that need to be kept over time?

Context:

You have considered using a *Summarizer*, but you need to keep a large amount of data. The amount of data you need to keep may vary based on the system configuration. Ideally, the number of samples would be a power of 10, but other exponent based representations will also work.

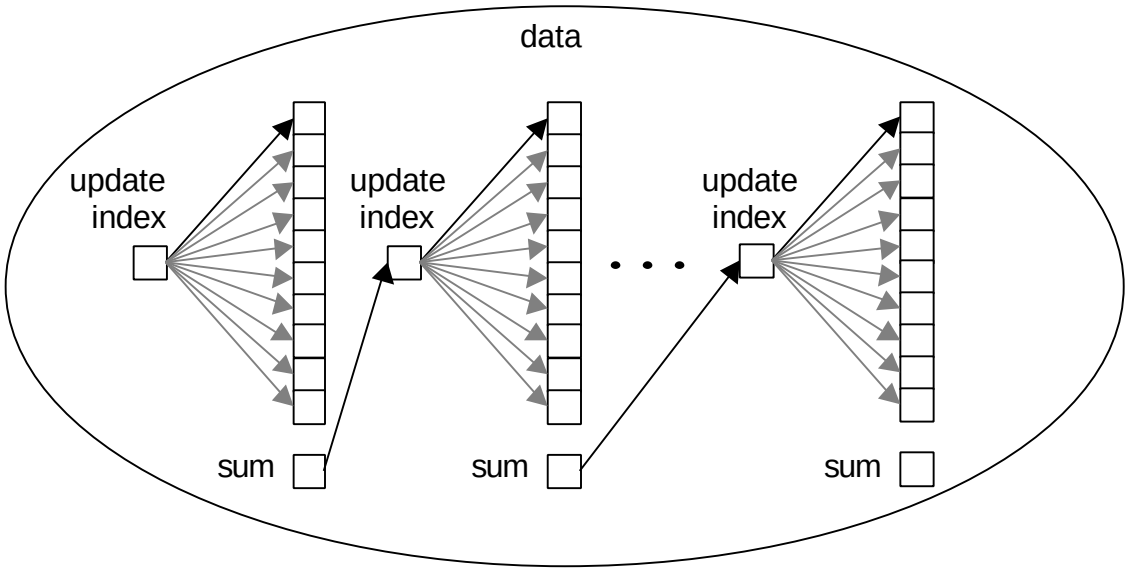
Forces:

- The memory needed to store the data can be very large. A storage method that minimizes the memory usage should be considered.
-

Solution:

Keep a set of nested *Summarizers*. In the case of representing powers of ten, each of the *Summarizers* should contain ten elements. One level of nesting is needed for each power of ten. For example, three levels would be needed to represent 1000 (10**3) samples of data. The sum of each nesting is stored as an element of the containing *Summarizer*. In the bottom most nesting, individual samples are stored. As the *Summarizer* wraps around, the next level *Summarizer* sample index is incremented. Care is needed when the a higher level *Summarizer* wraps around, as the data that is being replaced at the bottom level is the newest set of samples, not the oldest. This can be compensated for by spreading the value of the higher level Summarizer into the lower level *Summarizer*. This won't be 100% accurate, but will smooth out the values that had been collected over the data being replaced.

The size of the data type representing each sample can be limited to the maximum expected value, thus saving more space. Note, though, that each succeeding level of *Summarizer* needs to have the capacity to hold the sum of the maximum values of the samples of the *Nested Summarizer*.



Resulting Context:

If a 100% accurate solution is needed, this solution will not work, and a very large *Summarizer* must be used instead.

There are more details to this solution than are provided here. The implementation greatly depends on whether a functional or object oriented implementation is being used. This solution lends itself to dynamically changing the number of samples being represented, for example, changing from 10**3 to 10**4, but the details for doing this are omitted at this time.

Rationale:

Collecting error data for very small rates results in a huge number of samples to be taken to provide a sliding window calculation with 100% accuracy. Synthesizing the calculations by storing the intermediate sums results in less data being used and greatly speeds the calculation of the sum. Though the synthesized sum of data is not 100% accurate all of the time, it is better than 99% accurate all of the time for data that is spread out over time.

However, this slight lack of accuracy is more than compensated for in the amount of memory saved. A set of 1000 data samples can be represented with 30 elements. A set of 10,000 data samples can be represented with 40 elements. The saving dramatically increase as more data samples are represented.

References

[PLOPD2] M. Adams, J. O. Coplien, R. Gamoke, R. Hanmer, F. Keeve, and K. Nicodemus. "Fault-Tolerant Telecommunication System Patterns." In J. Vlissides, J. O. Coplien, and N. Kerth (eds.), Pattern Languages of Program Design 2. Reading, MA: Addison-Wesley, 1996: 555-556.

Acknowledgments

The author is grateful to a number of people in the telecommunications industry who have reviewed this pattern language and have validated the contained patterns:

- Richard Fackrell of AG Communication Systems who worked with me on the Traffic Data Administration subsystem of the GTD-5, where these pattern were originally learned and discovered. Rich was instrumental in reviewing and validating these patterns.

- Karen Grover and Pratima Shah who co-authored *Instant Event Snapshot* in the original AG Communication Systems Writers Workshop. Influence from this pattern can be found in portions of the pattern language.

- Bob Hanmer and Frank Buschmann for validating the use of these patterns in their respective companies.

- Frank Buschmann for shepherding this submission for PLoP'98.