

# **DISTRIBUTED SUPPORT**

**NEXT GENERATION DEVICE HEALTHCARE**

## Synopsis

Device support is inherently a local process, with a central database that can be loosely coupled. Distributed processing lends itself well to this, by using relatively powerful local processing.

## Precedent

Distributed computing seems to be a fashion industry. In the early days of computing, hardware was too expensive to use anything other than dedicated computing resources. We would submit our deck of punched cards, and for a few minutes, the computer was completely ours. It wasn't particularly convenient for us, but it kept the expensive computer busy all the time.

When the price of hardware started to fall a bit, it made more sense to have timesharing systems, where we had "dumb terminals" and used a single machine with an operating system that gave us the illusion that we had full access to the computer, even though we were actually sharing it with others. The computer was used a bit less than before, but it was much more efficient with our time, so the costs worked out better.

When the price of hardware fell even further, it made more sense to have dedicated workstations and personal computers for everyone. The abundance of memory and computing power made things like graphical user interfaces possible. Everything was designed around our convenience.

Next, the price of networking fell, and as a result of emerging networking standards, the "wired" computer became the norm. This caused three major changes:

1. the birth of distributed applications with remote data (email, browsing, games, telephony),
2. the need for central data repositories, and the corresponding birth of the server, and
3. an explosion of software applications (both wanted and unwanted) that took advantage of the convenient distribution mechanism.

Now an interesting trend is happening. The price of hardware continues to fall, but the application explosion is driving up the price of supporting that hardware. It has now become a viable option to reduce support costs by moving them to one place (an "applications server") and use "thin clients" locally. This is essentially going back to the timesharing paradigm, but with somewhat "smarter" terminals.

However, the attraction of distributed computing is strong. Many so-called "thin client" architectures use significant local processing. The value of fast response times and local autonomy is just too large to ignore. The real truth is that there are advantages to both kinds of architecture. Centralized data and distributed processing both play a role in dynamic, useful applications. The real solution is to bring down support costs by using Distributed Support, not mitigate them by trying to move everything to a central server.

There is an existing precedent for this kind of distributed support. A "self-help" support system gives end users access to a support knowledge base, and helps them diagnose and resolve support issues in a distributed fashion. Although this is not really a workable model for general support, it does demonstrate the viability of a distributed support model, and indicates its power in lowering support costs. In this model, the lower costs are a result of transferring cost and inconvenience directly to the end user, so this model doesn't work well in the general case.

## Detail

A key feature of the local processing architecture is to provide these required elements in the client software:

- the depth needed to handle complex diagnosis that requires filtering and using multiple sources of data,
- the power to apply detailed resolutions that utilize multiple low-level functions of the operating system, and
- the capability to incorporate relatively sophisticated logic in driving this process

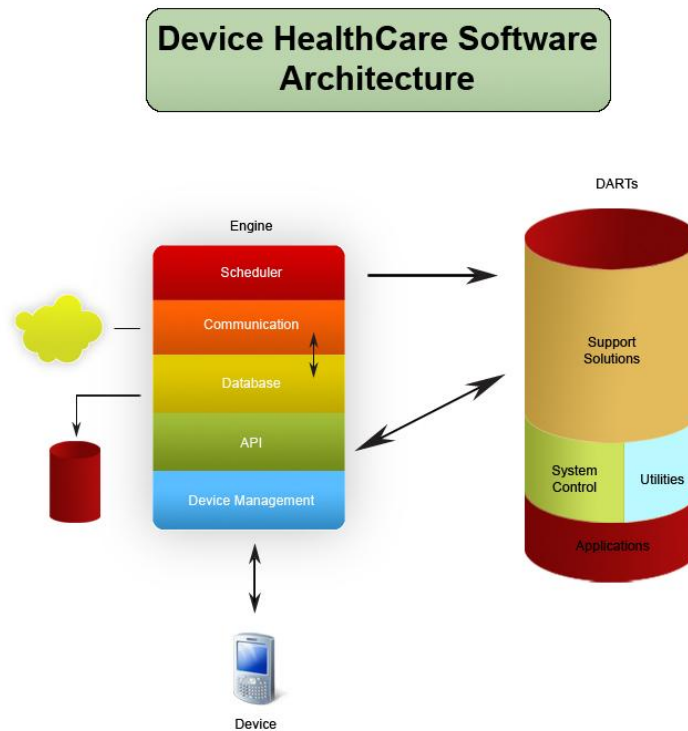
while keeping the complexity of the implementation at a reasonable level. The main way this is done is with a "divide and conquer" approach that maintains strict modularity.

The main modular division of the client is into two parts:

- **Engine:** this is the piece that provides the interface to the operating system, manages all communications to the server and other clients, and manages all scheduling and timing.
- **DARTs:** (Diagnosis, Analysis, and Resolution Tools) this is the collection of issue-specific solutions that represent the known detection, diagnosis, and resolution information of the universe of support issues.

DARTs do not directly interface to the system. Instead, they use a well-defined utility interface provided by the engine. Similarly, the engine has no issue-specific support knowledge. Instead, it just provides a framework for the DARTs to implement operations required for the support process.

The second modular division of the client is into the individual DARTs themselves. Every DART is completely independent of every other DART. Even though the engine is highly multi-threaded and multiple DARTs can be running at the same time, there is never any need to worry about synchronization between two DARTs because they are not interdependent. This also relieves the DART designer, and the engine structure, the burden of issues concerning the ordering of DARTs, their initialization, and their dependencies. DARTs can be loaded, unloaded, scheduled, and triggered using relatively simple mechanisms. Each DART can be debugged and tested in isolation with a high degree of confidence that it will operate correctly in a production environment.



Even though DARTs are independent, they can be related. As a result, there can be collections of common routines that are used by more than one DART. These are called PROCs and are simply a convenience to avoid duplicating parts of one DART in another. They are only required to add common code that is issue-specific, since this kind of code is not allowed in the engine.

There are actually several different classes of DARTs. These classes are not rigidly enforced, and are more of a notational convenience for understanding the structure of the client.

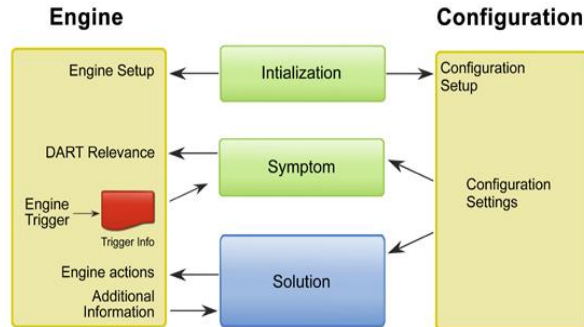
- **Issue:** this is the main type of DART, the one that addresses a specific support issue.
- **System:** Small collections of DARTs are actually used to implement certain client functions, such as synchronizing the configuration of database tables. These DARTs don't actually implement issue-specific support solutions, but the functions are implemented as DARTs because it is much more convenient to do certain tasks with a DART than inside the engine:
  1. run multi-threaded
  2. schedule

3. update
  4. configure
- **Utility:** this type of DART provides a general mechanism to address new support issues for which no specific DART exists yet. An example is a DART that can run a command line to use a built-in operating system utility to make a configuration change.
  - **Application:** this type of DART implements an entire feature in the product. An example is the DART that is used to implement Microsoft Update Management. DARTs are powerful and flexible enough to use for features like this. They provide a collection of features that can ease the transition from a "managed service" platform to a true Device HealthCare paradigm while providing a familiar environment for customers.

The third modular division of the client is the internal structure of a DART

- DARTs are divided into three parts.
  1. **Initialization:** Here, the DART tells the engine how it can be configured, what conditions should trigger it, and what kind of local storage it needs.
  2. **Symptom:** The engine runs this section of the DART when it is triggered; giving it detailed information about the trigger. The job of the symptom is to provide any more complex decision logic needed to decide whether or not the conditions for running the DART apply, and return that decision to the engine.
  3. **Solution:** If the symptom determines that the DART should run, the engine runs this section of the DART. The solution implements the action that the DART takes.

### Device HealthCARE DART Architecture



- DARTs are triggered and scheduled by the engine. During its initialization, the DART tells the engine the conditions under which it should be triggered. These conditions are defined by the rich set of underlying events used by the engine, and can be combined and filtered to make as specific a trigger as desired. The resulting trigger specification is processed efficiently by the engine, and when a DART is triggered, the engine schedules it in the DART queue along with information about the triggering conditions.
- The DART scheduler runs DARTs as multiple threads and manages the load to minimize the impact on the system. The engine also coordinates access to shared resources by multiple DARTs, synchronizing them so that the DART developers need not worry about conflicts or race conditions.
- DARTs can have parameters that control their operation. An example is a DART that acts based on a threshold on a free resource, where the threshold has a default value but can be configured. These configurations are managed by the engine, and can be configured for groups of devices on the server. The DART always has access to the configuration information as it runs. This mechanism is described in more detail in Pantographic Support.

Data modularity of the client is achieved through a database organization. The database is implemented by a small but powerful local SQL-based implementation. The data consistency is maintained by sharing the database schemas across the client and server.

This modular architecture for the client gives it significant local processing power without giving it a large footprint on local resources. The organization of the engine as reusable functional units makes it small and efficient. The organization of DARTs as independent triggered solutions makes them usable on an as-needed basis. The organization of the entire architecture as event-based rather than polled makes it low overhead because it is only active when it is actually needed.

## Benefit

The implementation of Distributed Support that is an integral part of Device HealthCare provides many benefits to any support organization.

- The flexibility and power of the modular client architecture provides the basis for a number of key advantages.
  1. Scaling to large numbers of devices can happen without overwhelming a shared central resource. Moving the bulk of the issue processing to the device itself means that the resources for addressing support issues increase at the same rate as the number of supported devices.
  2. At the same time, scaling to large numbers of devices can happen without overwhelming communication bandwidth. Moving the support process to the device where the issue actually occurs dramatically reduces the data requirements to a central resource.
  3. The client implements deep and detailed diagnosis capabilities. This is because a large volume of information can be locally processed intelligently instead of transferring all the data to a central resource for processing.
  4. Similarly, the client implements much more effective resolution of many different types of issues because it manages the device at the operating system level.
  5. The local intelligence of the client greatly increases the complexity of issues that can be addressed effectively. In addition, the exception management capabilities built into the engine allows easy diagnosis of unexpected circumstances that arise in the field that interfere with the proper operation of solutions.
  6. Issue management is quick and responsive, because the process from detection through diagnosis and resolution happens locally instead of relying on a central resource that may be intermittent or slow.
  7. The reduced data flow between the local client and central server also reduces the impact of privacy and security issues involved in supporting end user devices.
- The DART architecture allows the service offering to be customized to a particular need. In order to support a single product, the DARTs supplied with the engine can be tailored to meet the specialized needs of just that product.
- The common database architecture shared between the client and server provides a platform that is open and accessible to external interfaces.
- The modular architecture of the client gives it a small size and low overhead that can be deployed in situations where computing resources might otherwise cause difficulty in providing support solutions.

## Uniqueness

Simply put, no other offering has this kind of powerful and sophisticated software running on the supported device. Some other simple architecture in use is:

- The "client-less" architecture, where there is no client at all, and all access to the supported device is through externally provided interfaces. Vendors of these systems actually claim this as an advantage, since "no installation is needed". In fact, in order to get this one-time "advantage", you pay over and over by living with the inadequate and inconsistent information and control provided through the external interface.
- The "monitoring" client, which is a collection of stubs that collect a fixed set of information in a polled mode, and constantly transfer this information to a central resource for processing. This works fine for a "monitoring" solution, but has scaling problems and cannot possibly support the more sophisticated Device HealthCare paradigm. It is difficult to extend simple software architecture like this to a more complex one without just discarding it and starting over.
- A client that is really just a collection of independent scripts or executables that are run, possibly on a timed basis, by a simple program. These scripts can do a certain amount of work, but since there is no real framework for triggering and managing them, and providing common services to them, the whole structure begins to fall apart

with 100 or so of these scripts. In other words, the system has scalability issues with the size of the problem set rather than the number of devices.

Here are some key indicators to look for that separate a true Device HealthCare engine from an offering that cannot really handle the full range of demands.

- Does it have an architecture that separates the system level layer from the issue-specific solution mechanism? Without this strict modularity, handling the ever-growing number of support issues will soon overwhelm the software development process for the product.
- Is there any evidence that the same software architecture is flexible enough to implement complex applications? If "features" like patch management, backup, remote control, or security are implemented as "add-ons" that need to be developed separately outside the framework of the main product, then it is unlikely that the software architecture of the main product will scale to support the development needed to cover the requirements of the growing universe of support issues.
- How big is the installation for the client? How much room does it take once it is installed? If it is already a huge piece of "bloatware" then it will never survive the growth required to manage future support needs without causing interference on the supported devices.
- Does the software rely mainly on polling, or does it primarily use event architecture? Polled implementations will, by definition, use more and more of the resources of the supported devices as the number of issues it handles increases. At some point, it will simply stop being usable.