

RESTful CFDP: Managing GDS Complexity with Microservices

Joshua S. Choi

NASA Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California, 91109, United States of America

NASA's Advanced Multi-Mission Operations System (AMMOS) is currently adding capability to support the CCSDS File Delivery Protocol (CFDP). This feature is being added as part of the AMMOS Mission Data Processing and Control System (AMPCS). In order to address the system's increasing complexity, AMPCS has recently been re-architected to break down its monolithic applications into smaller, individually deployable microservices. The CFDP capability is the first new AMPCS feature to leverage this new architecture. The CFDP microservice provides a web-based Representational State Transfer (REST) application programming interface (API) for complete monitor and control of its operations, and this enables it to be decoupled from other AMPCS microservices. This also results in better scalability for redundancy and load balancing. AMPCS's CFDP microservice is designed to support generic CFDP operations, agnostic to AMPCS's legacy concept of Downlink Products. An optional runtime plug-in allows the CFDP microservice to simulate CFDP artifacts as Downlink Products. Applying the microservices software architecture pattern both in the latest release of AMPCS and in providing the new CFDP capability has resulted in a more flexible system with improved extensibility and maintainability. System complexity has also become more manageable.

I. Nomenclature

ACK	=	positive acknowledgment
AMMOS	=	Advanced Multi-Mission Operations System
AMPCS	=	AMMOS Mission Data Processing and Control System
API	=	application programming interface
ATLO	=	Assembly, Test and Launch Operations
<i>bps</i>	=	bits per second
CCSDS	=	Consultative Committee for Space Data Systems
CFDP	=	CCSDS File Delivery Protocol
CLTU	=	Communications Link Transmission Unit
CAM	=	Common Access Manager
CPD	=	Command Preparation and Delivery
GDS	=	ground data system
GSFC	=	Goddard Space Flight Center
HTTP	=	Hypertext Transfer Protocol
HTTPS	=	Hypertext Transfer Protocol Secure
ID	=	identifier
JDBC	=	Java Database Connectivity
JMS	=	Java Message Service
JSON	=	JavaScript Object Notation
<i>kbps</i>	=	kilobits per second
<i>Mbps</i>	=	Megabits per second
MSG	=	messaging
NAK	=	negative acknowledgment

¹ Engineering Applications Software Engineer, Mission Control Information Systems Group.

NASA	=	National Aeronautics and Space Administration
PDU	=	protocol data unit
SLE	=	Space Link Extension
TCP	=	Transmission Control Protocol
TC	=	telecommand
TM	=	telemetry
UI	=	user interface
WSTS	=	WorkStation TestSet

II. Introduction

The CCSDS File Delivery Protocol (CFDP) is a flexible and efficient file transfer protocol. Its primary target use is to enable the ground data system (GDS) to copy files to and from a spacecraft's mass memory. The protocol is a standard both developed and agreed upon by the international body of space organizations. By means of CFDP, various GDSs and spacecrafts implemented by different organizations can have a common, interoperable way to transfer files between flight and ground elements. The protocol also provides an optional service class that enables the files to be transferred reliably. Similar to the Internet's Transmission Control Protocol (TCP), this CFDP service class uses acknowledgments, timers, error-checks, and selective retransmissions of its protocol data units (PDUs) to provide this guarantee of delivery. Organizations designing and implementing systems based on CFDP can therefore save much work, and they can instead focus on the higher-level application logic. This also may provide simpler operational scenarios. As it becomes more commonplace for spacecrafts to use mass memory and very large data files, we can expect that missions will increasingly develop their concept of operations around CFDP. In fact, many successful missions have already flown with the CFDP capability on board, such as NASA's Juno, Mars Reconnaissance Orbiter, and MESSENGER missions, among others.

AMMOS Mission Data Processing and Control System (AMPCS) is a software application that is part of the greater Advanced Multi-Mission Operations System (AMMOS). AMMOS is one of NASA's primary mission operations GDS for robotic spacecraft missions. AMPCS, specifically, performs the following major functions: (1) Telemetry (TM) processing; (2) information monitoring, storage, and query; (3) automation support; and (4) commanding support. Work began in 2017, after an initial prototyping effort, to add a capability in AMPCS for supporting CFDP. This will allow AMPCS to be used as the GDS for some of the planned future missions that are expected to use CFDP on board, such as NASA's Europa Clipper mission.

AMPCS, which started out over a decade ago to be used as a simple, light-weight test tool during flight software development, has evolved and grown over the years to become a large and complex software system to handle many more use cases. Although its core functions are multi-mission, every mission that it was contracted to support had their own, unique set of requirements to which the system needed to adapt. It was also extended to support the missions beyond their Phase C activities but also all the way through Phase E: In addition to WorkStation TestSet (WSTS) environments, AMPCS is now used in mission testbeds, in Assembly, Test and Launch Operations (ATLO), and in mission operations. The different use cases and concepts of operations that AMPCS now supports is varied and large in number. This, however, has resulted in AMPCS turning into a large and complex software system. Just before the recent re-architecting, AMPCS consisted of over 8 million lines of code (primarily in Java, Python, and Extensible Markup Language, or XML). In terms of software architecture, it became monolithic. It was apparent that the original AMPCS design lacked a scalable framework to allow for an organized, structured extensions to its software. For example, many AMPCS applications had both their back-end logic and their front-end user interface (UI) implemented inside a single executable program. Classes from different Java projects were being cross-linked in various parts of the system, resulting in logical couplings that are very difficult to disentangle. Also, a number of developers joined and left the AMPCS team over the years, and each seemed to introduce new design patterns in the system, mainly due to the lack of an overarching, extensible framework to build their work upon. These and other issues gradually made AMPCS harder to extend and to maintain. This problem is not unique to AMPCS, however. Many software projects seem to face this all-too-common problem of their architecture eventually becoming monolithic and complex after many years of continuous development.

Adding new capabilities to AMPCS, such as the support for CFDP, was going to be a technical challenge with the existing architecture. In order to solve this problem and to help ensure continued usefulness of AMPCS in the future, it was therefore decided that the system must undergo a major change. Informed by a recent trend that emerged within the computing industry, the decision was made to re-architect AMPCS into a more *microservice* pattern, and the new CFDP capability would leverage this new architecture.

When this work began, the expectation was that the new system will have a framework in place that makes the system complexity more manageable. AMPCS would become a more agile system, one that can be easily extended to provide new capabilities without significantly increasing its complexity. This paper discusses what this work involved and some of the results, primarily in the context of the system's new CFDP capability. This paper also includes a discussion of what the aforementioned microservice architecture pattern looks like and how it was applied to AMPCS. There is also a focused discussion on the new CFDP microservice, including some of the user and system considerations that needed to be taken. Readers who are responsible for developing or managing GDSs, as well as those who are interested in mission operations that involve CFDP, may very likely find the information presented in this paper relevant to their work.

III. Microservice Architecture

The microservice software architecture pattern is a way of organizing an entire software system into an ecosystem of small applications—or *microservices*—that each do one specific thing very well. A microservice is independent of other parts in the system. Typically, a microservice will own its own data by having its own database. Microservices communicate with each other primarily via application programming interfaces (APIs).

This microservice pattern helps teams improve upon the way how software development and maintenance for large systems are typically done. Monolithic applications usually have code repositories that are also monolithic. With these huge code repositories, anytime a developer makes one change to the main code base, there is a risk of that change affecting other parts in the system, introducing unintended side effects. This is a common pain point for many teams that own large applications. With microservices, however, each microservice can be maintained in its own, smaller code repository. This helps isolate the effects of code changes that are made to some portion of the system, significantly reducing—if not preventing—unintended side-effects from being introduced. In the long run, systems based on microservices can be easier to maintain. Monolithic code repositories can be abandoned. Furthermore, the microservice pattern helps decouple work itself. Developers focusing on specific specialties or concerns can mostly work independently of each other, and in parallel, because the microservices that they own are themselves decoupled. They can also work, for the most part, without the constant worry about how their piece affects the whole system. With large systems, all of this helps boost productivity and allows the development teams to be more agile.

In deployment, microservice-based applications tend to be more dynamic and flexible than monoliths. Based on the user's operational needs, deployments can be scaled up or down to meet target performance and availability objectives. Deployments can also leave out some of the system's microservices that the user does not need. Designing monolithic applications that provide similar levels of robustness and flexibility is very difficult, and it often makes the system complexity unmanageable.

During the past several years, a number of leading tech companies such as Netflix and Amazon shifted from monolithic applications to using microservices. For these companies, the main driver behind this change is the ever-increasing availability of cloud computing resources. Companies are moving away from maintaining their own data centers and more toward hosting their primary business applications on the cloud. Cloud service resources can be highly elastic, which is a great fit for running the small and dynamically-deployable microservices. Fig. 1 depicts a conceptual on-demand video streaming service based on the microservice architecture. In the figure, each circle represents a microservice, some with its own database or local file store. These microservices communicate with each other via APIs. Some microservices, such as the ones that perform analytics, consume user-triggered and system-internal events via the Message/Event Bus. (Systems that handle very large amounts of real-time events will often split the communication bus into two or more, one to handle all incoming data in their raw form and another to carry a lower-volume, transformed set.) Some types of data may be made commonly available to different microservices, such as the very large media files (inside Raw Media Stores) that need to be streamed to the users (via the Content Delivery Service), after being encoded and having their manifests generated.

IV. AMPCS Re-Architecture

AMPCS's previous architecture had a number of characteristics that were not problematic for its original use, but gradually had become liabilities, as the system evolved to meet greater demands. As mentioned in the introduction, AMPCS applications had interactive UIs embedded in the very applications that the interfaces needed to monitor and control. This prevented users from being able to remotely control the applications unless they used a remote desktop solution such as Virtual Network Computing (VNC). Also, AMPCS applications generally lacked APIs that could allow external applications to control them, which would be crucial in enabling completely flexible, lights-out types of automation. (Note: To provide an automation solution despite the lack of such APIs, AMPCS provided—and will continue to provide—the MPCS Test Automation ToolKit, or MTAK, and AMPCS Utility Toolkit for Operations, or

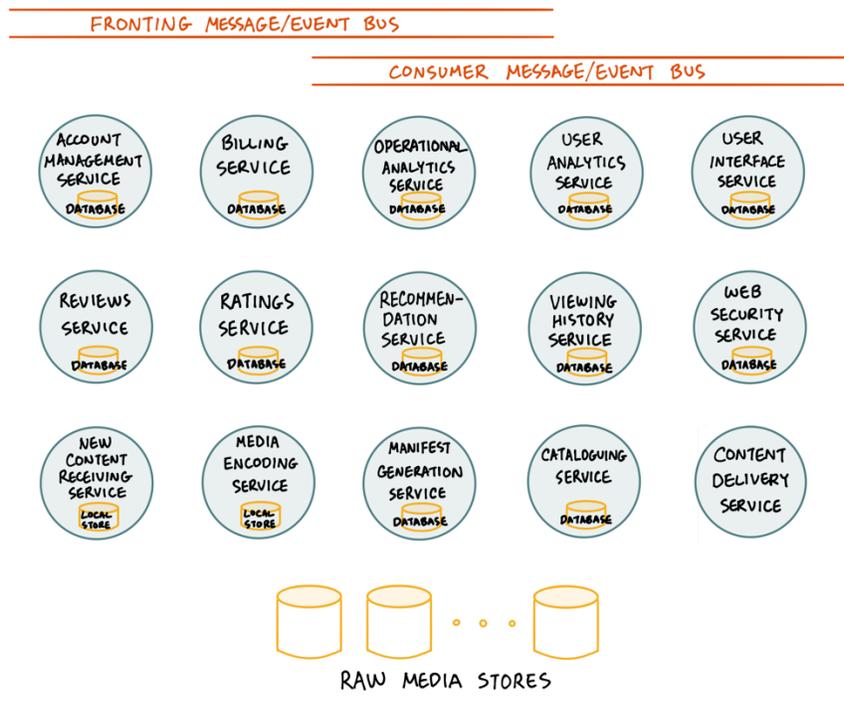


Fig. 1 An example architecture of an on-demand video streaming service based on the microservice pattern.

AUTO, for automating some set of telemetry and telecommanding functions in test venues and in operations, respectively.) AMPCS was also designed with a fixed, inflexible data processing chain, which was difficult to change. As AMPCS was tasked to support, over time, previously unplanned use cases and concepts of operations, this rigidity became a severe limitation. In addition, much of the data that passes between the processing elements in the chain were not being shared with external applications, which prevented GDS engineers and mission users from making use of that data for additional services. Another limitation was that, due to AMPCS's monolithic architecture, it was always being deployed with its full set of capabilities, despite the target mission requiring or requesting only a subset of them. For these customers, the unneeded capabilities were simply disabled as AMPCS was deployed, rather than the preferred way of having those capabilities not being deployed at all. These and other limitations needed to be addressed in AMPCS eventually.

During the past two years, the AMPCS team undertook a major re-architecting effort on the system in order to solve the aforementioned problems and to convert some of its applications into microservices. The new AMPCS, labeled Release 8, is now built on top of the Spring Framework, which is a Java application stack that helps enforce clear boundaries and good software organization. By using this framework and after much hard work by the developers, the globally-shared objects in AMPCS that caused dependency entanglements were finally replaced in the system, allowing its applications to be split into smaller microservices. The Spring Framework also provides native support for web applications. This proved valuable when converting some of the existing AMPCS applications into RESTful web services. A RESTful web service is a service that provides APIs over the Hypertext Transfer Protocol (HTTP), or its Secure version (HTTPS), and support the Representational State Transfer (REST) interaction style [1]. With RESTful APIs in place, the graphical user interfaces (GUIs) could now be separated from the back-end logic into separate applications. Also, new command-line tools were created to interact with the applications via their new REST APIs.

Some work is still ongoing, such as breaking apart the fixed, rigid data processing chains in AMPCS. In a later release, the single downlink TM application that handles everything from connecting to the TM source; to processing the transfer frame stream; to processing the packets extracted from those frames; to processing the Event Verification Records (EVRs), Engineering Housekeeping and Accountability (EHA) data, Product data parts, et cetera; to performing channel derivations; to triggering alarms; and other processing steps will be split apart into two or more microservices. For example, the current plan for Release 8.1 includes breaking up that downlink TM application into

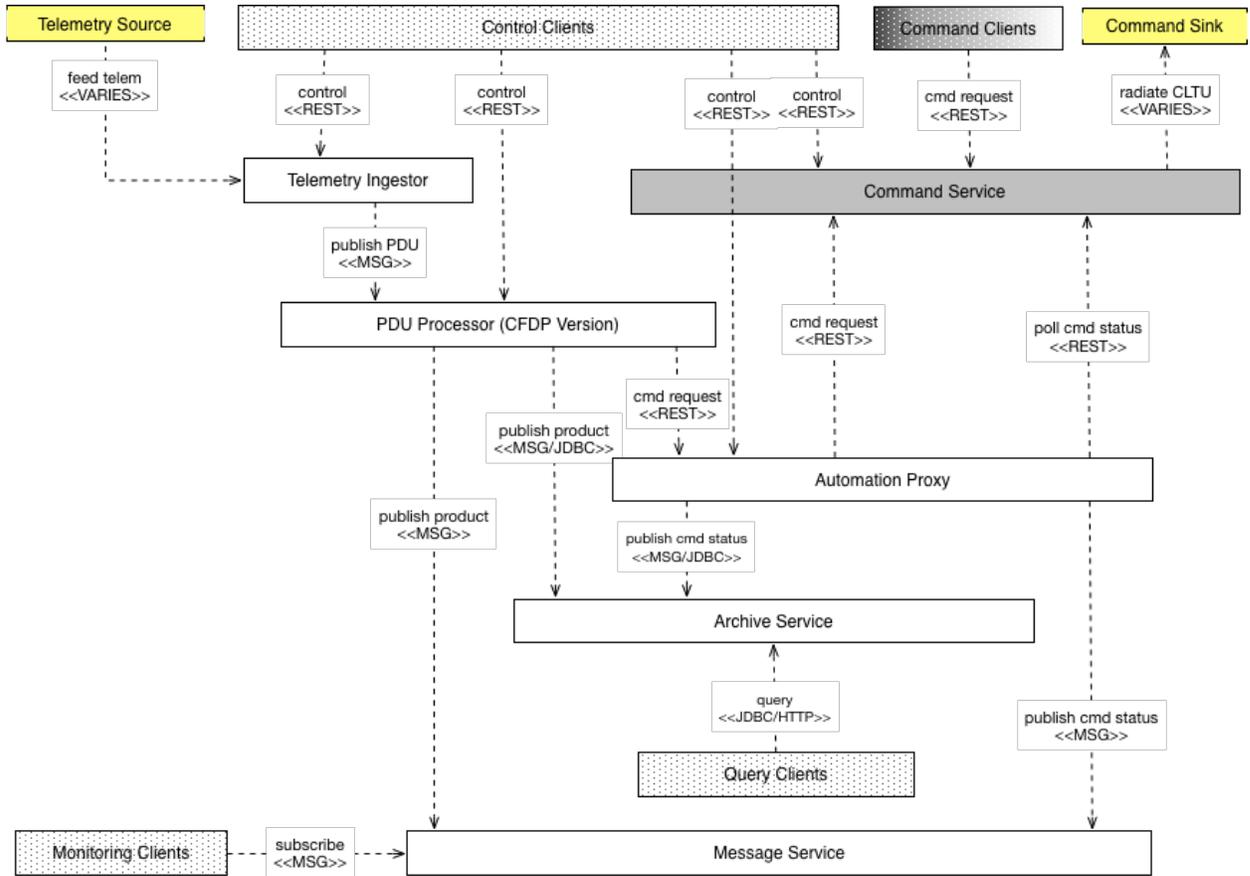


Fig. 2 The new architecture of AMPCS Release 8.

a Telemetry Ingestor microservice and a Telemetry Processor microservice. This split alone would provide the ability for multiple instances of the Telemetry Ingestor, each receiving data from different TM sources, to feed all the data to a single Telemetry Processor; or multiple Telemetry Processor instances can tap into the same Telemetry Ingestor feed and perform different types of processing, in order to balance the load.

Fig. 2 is a model of the re-architected AMPCS (Release 8). The white-filled boxes represent persistent AMPCS microservices, and the stippled boxes represent the transient, UI applications (grouped together by their types). The solid grey-filled box is an externally-provided microservice that AMPCS uses, and the stippled box with black-to-white gradient represents applications that have a mixture of both AMPCS and externally-provided elements. Yellow-filled boxes with open sides represent data source or sink external to AMPCS. The lines represent data interactions. `<<REST>>` represents interactions occurring via REST APIs (request/response paradigm), `<<MSG>>` represents interactions via the Java Message Service (JMS) (publish/subscribe paradigm), `<<MSG/JDBC>>` represents interactions where the initiator publishes the data via JMS and the responder uses the Java Database Connectivity (JDBC) standard to archive the data inside the AMPCS database, and lastly `<<JDBC/HTTP>>` represents the interaction where the initiator queries for the archived data in the database via JDBC over HTTP.

This re-architected AMPCS is a hybrid microservices system. For example, none of the individual microservices own their own data. All data are still pooled together under a single database managed by the Archive Service (albeit in separate tables). This is because, when transforming existing monolithic applications into microservices, it is usually more cost-effective to address the major “pain points” first, as the budget and time allows. As such, the objective of the AMPCS re-architecture effort is to address the major limitations that were mentioned earlier, by applying the microservices pattern where it can have the most impact, rather than a complete makeover of the system.

V. RESTful CFDP Microservice

AMPCS’s new CFDP capability is the first feature to make use of the new microservices architecture in Release 8. It seems fitting to provide this capability as a separate, independent microservice in the AMPCS ecosystem. Not all

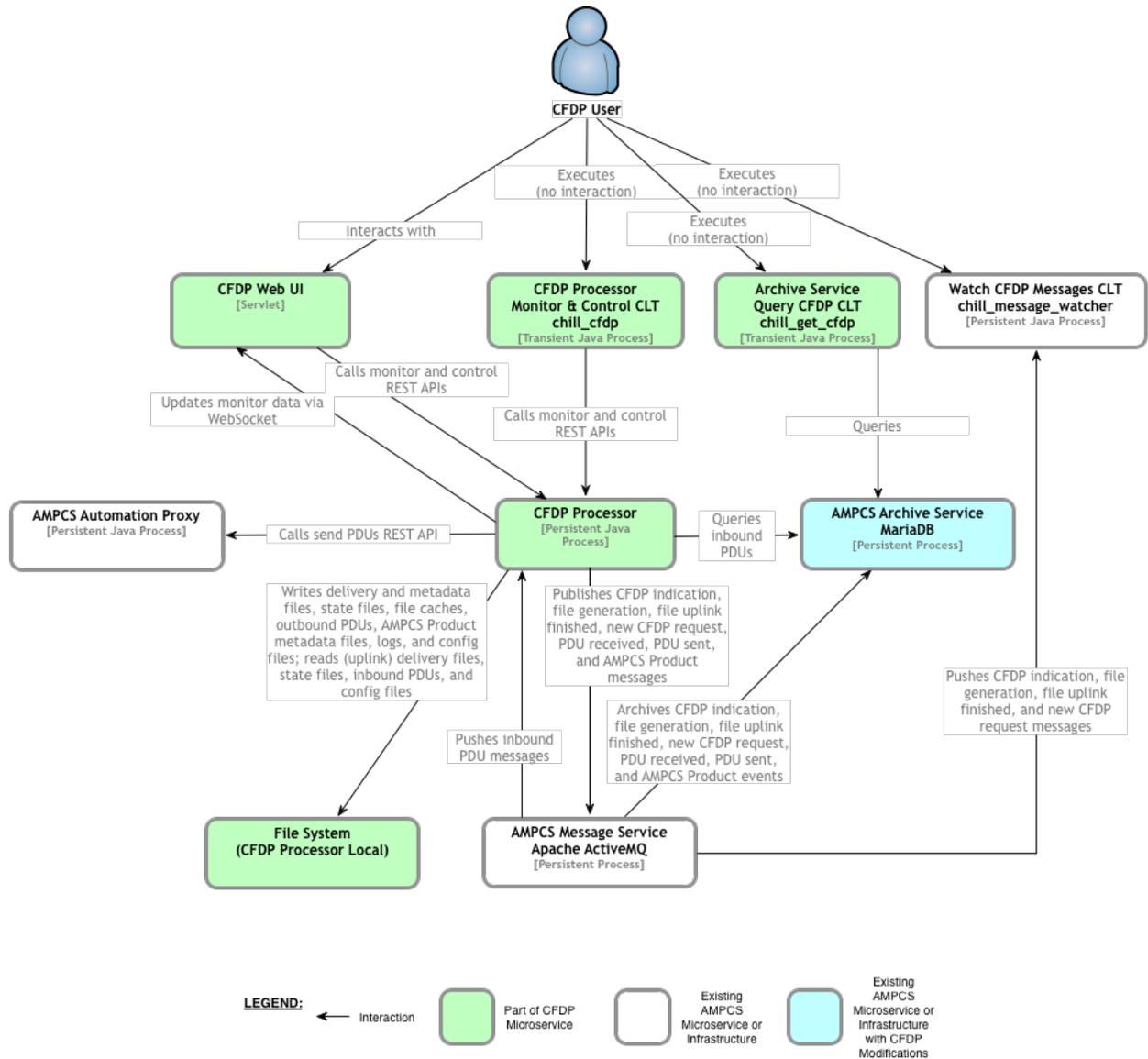
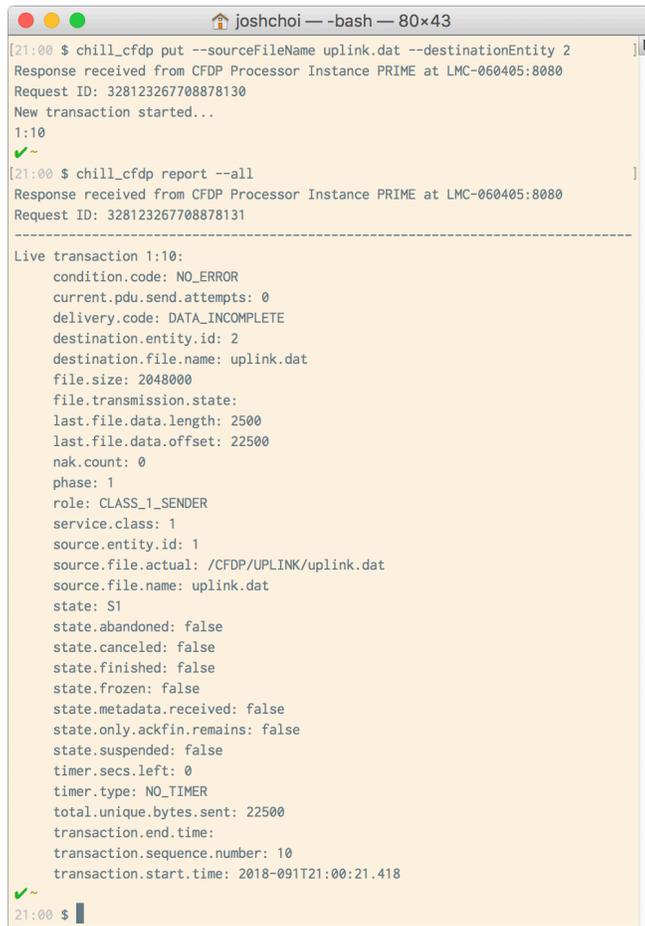


Fig. 3 The CFDP microservice model in AMPCS.

missions that use or will use AMPCS require CFDP. Also, CFDP maintains a complete separation of layers in the space communications protocol stack. For example, a CFDP PDU may occupy the payload field of a space packet (or a transfer frame), but the software that processes the space packet (or the transfer frame) can abstract the CFDP PDU as an arbitrary payload data. Likewise, the software that processes CFDP PDUs can abstract the space packet (or the transfer frame) as an arbitrary container. This allows the Telemetry Ingestor and the PDU Processor in Fig. 2 to be separated into independent microservices.

Fig. 3 shows the new AMPCS CFDP concept. The actual microservice that handles CFDP is labeled CFDP Processor. It exposes a REST API for monitor and control. A command-line tool, *chill_cfdp*, is provided for the user out-of-the-box so that they can easily interact with the CFDP Processor, in effect calling the latter's REST API under the hood. (Fig. 4 shows an example where this tool is used to start a new CFDP transaction and also to receive a report on all transactions being tracked in the system.) In a later release, a CFDP web user interface (web UI) will also be provided so that the users can perform the same monitor and control tasks graphically, using the same REST API. (The web UI will probably augment the REST API with an additional WebSocket protocol connection, as it is better suited for receiving continuous streams of monitor data with minimal overhead.)

A terminal window titled 'joshchoi -- -bash -- 80x43' showing the execution of the 'chill_cfdp' tool. The first command is 'chill_cfdp put --sourceFileName uplink.dat --destinationEntity 2', which returns 'Response received from CFDP Processor Instance PRIME at LMC-060405:8080', 'Request ID: 328123267708878130', and 'New transaction started...'. The second command is 'chill_cfdp report --all', which returns a detailed report for transaction 1:10. The report includes fields such as 'condition.code: NO_ERROR', 'current.pdu.send.attempts: 0', 'delivery.code: DATA_INCOMPLETE', 'destination.entity.id: 2', 'destination.file.name: uplink.dat', 'file.size: 2048000', 'file.transmission.state:', 'last.file.data.length: 2500', 'last.file.data.offset: 22500', 'nak.count: 0', 'phase: 1', 'role: CLASS_1_SENDER', 'service.class: 1', 'source.entity.id: 1', 'source.file.actual: /CFDP/UPLINK/uplink.dat', 'source.file.name: uplink.dat', 'state: S1', 'state.abandoned: false', 'state.canceled: false', 'state.finished: false', 'state.frozen: false', 'state.metadata.received: false', 'state.only.ackfin.remains: false', 'state.suspended: false', 'timer.secs.left: 0', 'timer.type: NO_TIMER', 'total.unique.bytes.sent: 22500', 'transaction.end.time:', 'transaction.sequence.number: 10', and 'transaction.start.time: 2018-091T21:00:21.418'.

```
[21:00] $ chill_cfdp put --sourceFileName uplink.dat --destinationEntity 2
Response received from CFDP Processor Instance PRIME at LMC-060405:8080
Request ID: 328123267708878130
New transaction started...
1:10
✓~
[21:00] $ chill_cfdp report --all
Response received from CFDP Processor Instance PRIME at LMC-060405:8080
Request ID: 328123267708878131
-----
Live transaction 1:10:
  condition.code: NO_ERROR
  current.pdu.send.attempts: 0
  delivery.code: DATA_INCOMPLETE
  destination.entity.id: 2
  destination.file.name: uplink.dat
  file.size: 2048000
  file.transmission.state:
  last.file.data.length: 2500
  last.file.data.offset: 22500
  nak.count: 0
  phase: 1
  role: CLASS_1_SENDER
  service.class: 1
  source.entity.id: 1
  source.file.actual: /CFDP/UPLINK/uplink.dat
  source.file.name: uplink.dat
  state: S1
  state.abandoned: false
  state.canceled: false
  state.finished: false
  state.frozen: false
  state.metadata.received: false
  state.only.ackfin.remains: false
  state.suspended: false
  timer.secs.left: 0
  timer.type: NO_TIMER
  total.unique.bytes.sent: 22500
  transaction.end.time:
  transaction.sequence.number: 10
  transaction.start.time: 2018-091T21:00:21.418
✓~
21:00] $
```

Fig. 4 Example of the CFDP command-line tool being used to start a new transaction and to view a report of all transactions.

REST APIs play an important role in the new microservice-based AMPCS. It provides a very flexible and robust mechanism for the microservices to communicate with each other. In simple terms, REST interaction involves a client entity and a server entity, and the client entity sends requests to the server entity, while the server entity responds back to the client for each of those requests. Each request specifies an action (e.g. GET) and an address to the resource on the server entity that the action should be applied to (e.g. <http://server/employees/john-doe/phone-number>). The specifiable actions are limited to the vocabulary made available by the HTTP request methods [2], but they are sufficient to cover all Create, Read, Update and Delete (CRUD) operations needed for manipulating resources. Because making REST API calls are basically a matter of issuing HTTP requests, a plethora of user and system tools, such as cURL, GNU Wget, Postman, nc, telnet or even a web browser, can be used as a REST API client.

Fig. 5 shows an example of a CFDP client application (e.g. CFDP command-line tool) initiating a CFDP Put request on the CFDP Processor via the REST API, namely by issuing a HTTP PUT request. The CFDP Put request and the HTTP PUT request are used together in this particular example, but the two are not to be confused with each other. Before this interaction can begin, the CFDP Processor needs to have the endpoint for CFDP Put requests mapped to the Uniform Resource Identifier (URI) `/cfdp/action/put`. In the example, the CFDP client also includes an optional key-value parameter in its HTTP request, `Content-Type: application/json`, which informs the server that the request's body contains data that is formatted in the JavaScript Object Notation (JSON), intended for the CFDP Processor application. Upon processing the HTTP request (i.e. the REST API call), the CFDP Processor can send a response back to the client with a status code [2]. In this case, the code is `200 OK`, which signifies that the CFDP Processor has successfully completed the request. In the example, the response also includes a body, in which a JSON data object

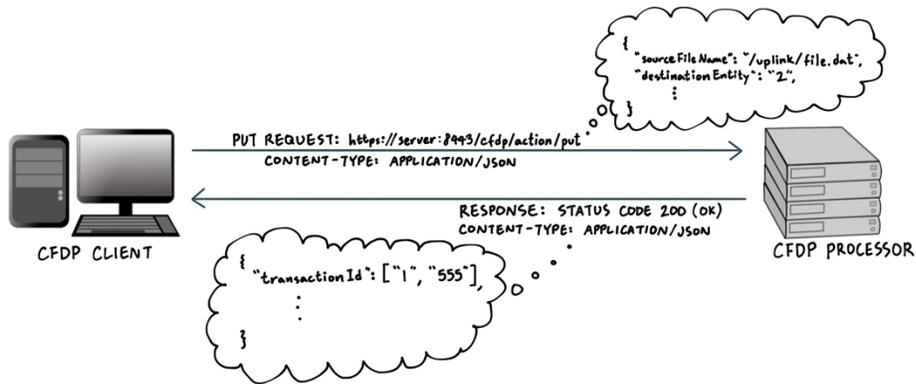


Fig. 5 Example REST interaction between a CFDP client and the CFDP Processor.

contains the newly created CFDP transaction's sequence number (555) as well as the source CFDP entity identifier (ID) used by the local CFDP Processor (1).

The CFDP Processor is a Spring Boot application, and so it is also built on top of the Spring Framework. Spring Boot provides a useful feature where, via a simple configuration, an Apache Tomcat® web container can be embedded inside the application. The CFDP Processor uses this feature, and all of its REST API endpoints are served by this embedded container, and so there is no need to deploy the microservice on a separate web server.

Internally, the CFDP Processor uses NASA's Java CFDP Software Library developed at the Goddard Space Flight Center (GSFC) for all low-level protocol handling. The Java library is a port of the C-based version of the same library, also developed and maintained by GSFC, and is considered reliable. (The C version has been used in the flight software of the Global Precipitation Measurement [3], Neutron Star Interior Composition Explorer, and Global Ecosystem Dynamics Investigation missions.) Some of CFDP Processor's REST API endpoints map directly to the user request API calls provided by the library (e.g. CFDP Put).

The CFDP Processor and its UI applications can be deployed independently of the rest of AMPCS. When no other AMPCS services are available, the CFDP Processor can read incoming PDUs from local files and also write outgoing PDUs as files. This may be fine for testing, but for a full, end-to-end CFDP-based operations, CFDP Processor depends on a running instance of the AMPCS Message Service (JMS bus) to subscribe to incoming PDU messages; it also depends on a running instance of the AMPCS Automation Proxy microservice to send outgoing PDUs to (transmitted via REST). The producer of the incoming PDU messages would be the AMPCS Telemetry Ingestor. The Telemetry Ingestor publishes the messages as it extracts the CFDP PDUs from a downlink stream of space packets (or transfer frames). In any case, the CFDP Processor and its UI applications can be deployed on hosts (or virtual machines) separate from the rest of AMPCS (and even separate from each other) or deployed altogether on a single computer.

When the Message Service is available, the CFDP Processor also publishes events to allow the users and external applications (e.g. automation software) to continuously monitor the CFDP operations in real-time. Events that the CFDP Processor publishes include CFDP Indications (e.g. Transaction-Finished), downlink file generation (which can occur before the transaction itself finishes), file uplink finished (also can occur before the transaction ends), user request received, PDU received, PDU sent, et cetera. When the AMPCS Archive Service is available, these events are also recorded inside the AMPCS database. Users and other applications can then later query past CFDP Processor events for analysis. Optionally, the CFDP Processor can archive the raw PDU data that it receives and generates into the database. A command-line tool is provided for the users to query these events and data from the database (*chill_get_cfdp*).

The CFDP Processor also writes metadata files, also known as transaction logs, to the file system. Each of these files contain information about a particular CFDP transaction, such as whether or not the received file is corrupt, the transaction's start and end times, source and destination entity IDs, transaction sequence number, bytes (or octets) transferred, and optionally, a list of all the PDUs exchanged during the transaction, each with their unique ID (generated and tagged by the CFDP Processor) and metadata. That last piece of information is useful for troubleshooting and for general data accountability.

VI. User Experience Considerations

The design approach taken with the AMPCS CFDP microservice is to prioritize providing users with the core functionality and use cases defined by the CFDP standard and not introduce AMPCS or mission customizations as best as possible. The goal is to allow any user with a working knowledge of CFDP to know how to use the AMPCS CFDP feature right out of the box—regardless of his knowledge of or experience level with AMPCS. However, we must also consider the group of mission users who are already accustomed to AMPCS and its concepts. For example, mission GDS users with prior experience with AMPCS expect to see all data, both TM and telecommand (TC), to be tied to specific AMPCS “sessions.” Also, CFDP is seen by these users as replacing a legacy file transfer protocol (files are known as Downlink Products in this case), but they prefer that much of the same terminology and tools be carried over. Because these users have established concepts of operations, procedures, automation scripts, test cases, and so forth, that been developed during past missions, AMPCS will need to continue its support for the user tools that handle legacy Downlink Products so that missions can reuse much of their inherited artifacts.

In order to satisfy both types of users, each with different expectations on how CFDP should, in effect, “look and feel” in AMPCS, we applied a plug-in model approach to the CFDP microservice. This plug-in, which can be configured to load or be skipped when the CFDP Processor starts up, is responsible for consuming CFDP events and translating them into legacy Downlink Product events (e.g. Metadata Received Indication Message to Product Started Message), translating CFDP metadata files into AMPCS Downlink Product metadata files (specifically known as Product Earth Metadata, or EMD, files, which is a heritage from NASA’s Mars Science Laboratory mission), and inserting entries into AMPCS database’s Downlink Product tables (to allow the existing *chill_get_products* tool to work with CFDP files). Using this approach, users can continue to use Product-related tools and UIs, such as AMPCS’s Downlink Products Viewer (*chill_dp_view*). Fig. 6 shows this plug-in model in the CFDP microservice context.

VII. Concept of Operation

Fig. 7 shows a reference concept of operation using AMPCS with the new CFDP capability. In this example, AMPCS’s TM source is the NASA Deep Space Network’s (DSN) Space Link Extension (SLE) [4] Return Service, either the Return All Frames (RAF) or Return Channel Frames (RCF) service. AMPCS uses a proxy application, called the SLE Proxy, to communicate with SLE Return Services. SLE Proxy forwards incoming transfer frames, via socket, to the Telemetry Ingestor. The Telemetry Ingestor then processes the data, publishing CFDP PDU messages (or, in Release 8.1, space packet messages intended for a separate Telemetry Processor microservice). The CFDP Processor consumes these PDU messages. This is the return path. On the forward path, the CFDP Processor sends PDUs via REST API to the Automation Proxy microservice. The Automation Proxy then prepares, optionally aggregating, the PDUs (and also possibly interleaving them with other TC data), finally submitting them as generic TC data in radiation requests made to the Command Preparation and Delivery (CPD) subsystem via REST API. The

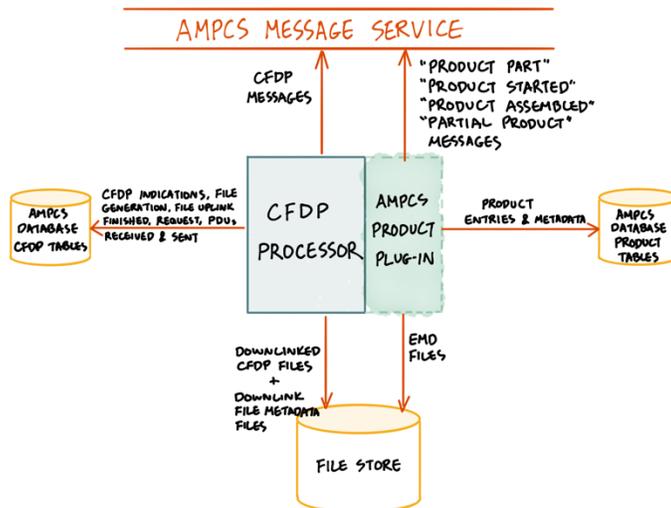


Fig. 6 Model of the AMPCS Downlink Product plug-in for backward compatibility.

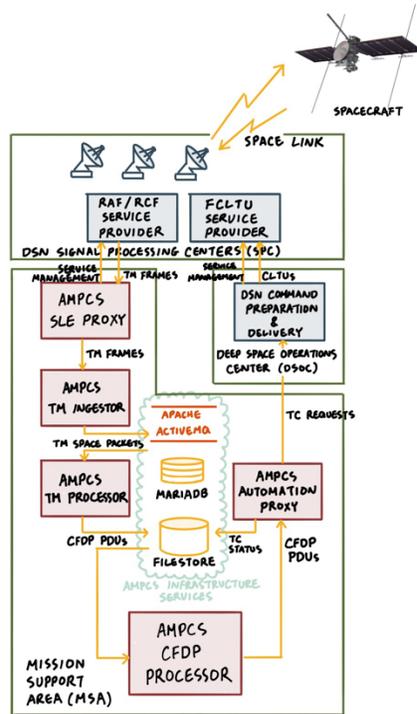


Fig. 7 Reference end-to-end concept of operation in AMPCS with the new CFDP capability in the loop.

CPD subsystem, in turn, uses the SLE Forward Communications Link Transmission Unit (Forward CLTU) service to radiate the data to the spacecraft.

VIII. Performance, Scalability, and Security

An important factor to consider with the new CFDP microservice is whether or not it can meet the mission test and operational performance requirements. The CFDP capability in AMPCS started out with the multi-mission requirements to support an incoming PDU throughput range of 10 bits per second (*bps*) to 28 megabits per second (*Mbps*) in operations and a range of 10 *bps* to 42 *Mbps* in test venues. For uplink, the requirements are 7.8125 *bps* to 1 *Mbps* in operations and 7.8125 *bps* to 2 *Mbps* in test venues. Recent internal testing showed that the CFDP microservice meets these requirements. In reality, however, the CFDP Processor can only process the TM data as fast as the upstream Telemetry Ingestor can make those data available, and currently that is where the performance bottleneck seems to lie. Nevertheless, the Telemetry Ingestor and the overall AMPCS continues meet the system performance requirements (500 kilobits per second, or *kbps*, of real-time downlink TM, 6 *Mbps* of non-real time TM, 64 *kbps* TC uplink in operations, and 1 *Mbps* TC uplink in test). A benefit of using microservices is that, because the individual services are decoupled, once we identify the specific microservice(s) that is (are) causing the bottleneck, we are able address the issue locally, such as by splitting up the microservice(s) into different smaller, better performant ones, all without introducing any sort of harmful effects on the rest of the system.

One of the benefits mentioned earlier about using microservices is that they can be made to scale up or down easier than monolithic applications. This is true also in the case of AMPCS's CFDP capability. For example, it is possible that a mission will require that AMPCS handle processing loads that a single CFDP Processor instance cannot keep up with. In this case, two or more CFDP Processor instances can be spawned on different computing resources to divide the work. This flexibility can be used to improve the system's availability as well. For redundancy, multiple CFDP Processor instances can run concurrently (all with a same local CFDP entity ID), processing the same downlink CFDP data, in case some of the hardware resources that they are hosted on experience failures. Fig. 8 shows an example of these configurations. The Telemetry Ingestor (or the Telemetry Processor in AMPCS Release 8.1, as depicted here) is itself a microservice, and multiple instances can simultaneously process the downlink TM stream(s). They then publish the CFDP PDUs received from the spacecraft(s) to the JMS bus. In the diagram, the CFDP Processor

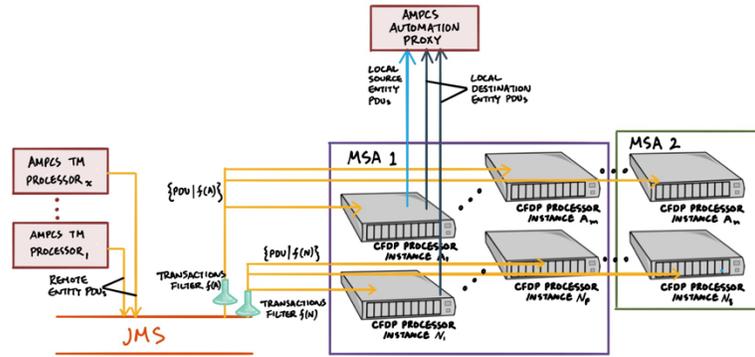


Fig. 8 An example scenario of the CFDP microservice being horizontally scaled to provide redundancy and load balancing.

nodes (top row) that subscribe only to the PDU messages which have been filtered by some parameter A all receive the same set of PDUs, and this provides redundancy. At the same time, another set of CFDP Processor nodes (bottom row) receives a different set of PDUs filtered by some parameter N . When these filters serve to partition the PDUs, this scheme provides load balancing. (JMS provides message filtering as one of its features, and the partitioning filter can be as simple as “only odd-numbered transaction sequence numbers.”) This strategy, however, does require some careful thought on how the outgoing PDUs can affect the remote entity. The situation is simpler when CFDP Service Class 1 (unacknowledged mode) is used, in which case any one or more of the CFDP Processor node(s) can be used for uplink transactions. In Class 2 (acknowledged mode), it is important for only one CFDP Processor node to generate the outgoing PDUs for a particular transaction (and the entire transaction must exclusively be owned by this node), otherwise the remote entity is at risk of receiving multiple of the same control PDUs (i.e. positive acknowledgments, or ACKs, negative acknowledgments, or NAKs, and so forth) and basically the transactions can fall apart in several ways.

In microservice architectures, there is increased need for vigilance with regard to security. Whereas in a monolithic application the interactions between modules are typically kept internal to the application process and therefore shielded from the outside world, in microservices, however, the REST APIs are at risk of being completely exposed. There are number of ways to mitigate this. For example, a private network infrastructure can be set up so that the REST API endpoints that are considered internal to the application can only be accessed from within the private network. Another approach—and this can be used complementarily to the previous example for even stronger level of security—is to secure all the REST API endpoints via some sort of authentication, authorization, and data privacy mechanisms. All of AMPCS microservices, including the CFDP Processor, apply this technique. AMMOS provides a security policy management and enforcement system called the Common Access Manager (CAM). All REST API requests in AMPCS are checked against the security policies in the CAM. These policy checks can be configured to occur at the web container-level, and so the Apache Tomcat instance (running embedded inside the AMPCS application) rejects outright any unauthorized calls to the service’s API, before passing them on to the proper resource controllers. This, together with a secure, institutional user directory system, exclusive use of HTTPS, and a private network, helps protect the microservices from outside attackers. Fig. 9 shows the general concept of this security mechanism at work for AMPCS CFDP.

IX. Conclusion

In this paper we discussed what the microservice software architecture pattern is and some of its advantages over monolithic applications. We also discussed how AMPCS has recently been re-architected to move toward this microservice architecture. CFDP is a robust file transfer protocol, and a growing number of missions and GDSs are adopting its use. This paper introduced the new CFDP capability in AMPCS, which has been designed as a RESTful microservice. We considered the engineering decisions that were made to implement this capability, as well as some of the results from that work. We also described some of the potential mission use cases, showing the flexibility benefits of applying the microservice pattern.

One important lesson that we learned from this work is that, changing the architecture from monolithic applications to microservices does not necessarily reduce the system’s complexity, but rather, it replaces certain types of complexity with different ones. For example, to relieve some of the pain points in AMPCS, the Spring Framework was incorporated, partly to help split up messy dependency entanglements and to remove the sharing of global objects.

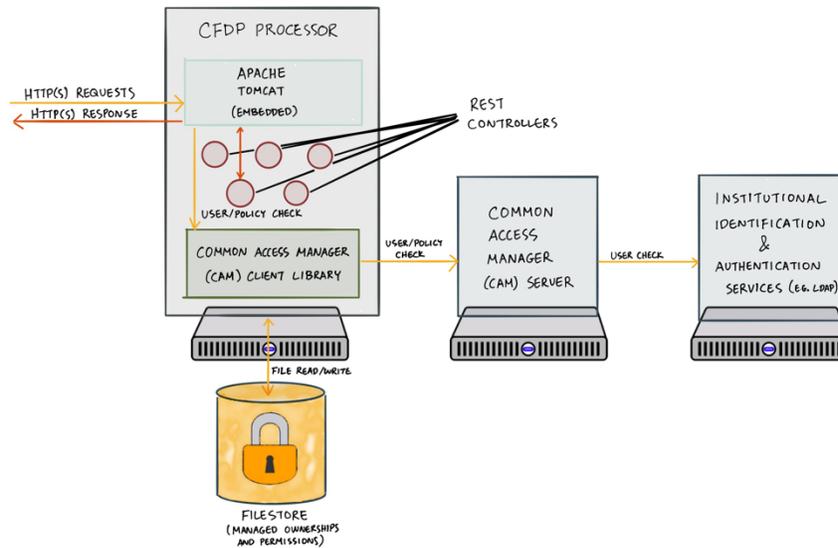


Fig. 9 Security mechanisms for the AMPCS CFDP microservice.

Eventually this resulted in the formation of clear boundaries in the system, and some of AMPCS's applications were then converted to leaner RESTful microservices. The CFDP Processor leverages this new pattern, and its introduction into the AMPCS ecosystem did not cause a significant increase in system complexity. On the other hand, AMPCS now depends on a new application stack (Spring) which it did not before, and this brings with it another set of challenges. The system is also now more modular and flexible, but we introduced new sets of REST APIs into the system, each of which now needs to be made secure.

The benefits of using the microservice pattern seems to outweigh its costs, however, when we take into consideration the system's improved extensibility and maintainability qualities. Adding new, core capabilities, such as CFDP, to a system that is as large as AMPCS, will have been far more complicated if it was done on the older, monolithic architecture. It would have introduced inadvertent side effects, undesirably modifying some of AMPCS's existing behavior, and thereby requiring costly remediation. Although currently all of AMPCS's code is maintained in a single, main repository, only a few code merge exercises were needed for the CFDP feature, mainly because the development of the microservice proceeded independently from rest of the system. Taking everything into account, the system complexity may not have been significantly reduced, but it seems to have become more manageable.

The new architectural pattern in AMPCS will also make it easier to add support for the Delay/Disruption Tolerant Networking (DTN) and other core services in the future. Going forward, such extensions to the system will not unduly increase its complexity. Also, AMPCS is now better suited to move on to the cloud platform thanks to the use of RESTful microservices. It may then be possible for missions to use this GDS without any on-premises hardware or facilities to host it, and this has the potential to decrease the missions' operational costs, as well as allowing the teams to become more location independent. These are just some of the likely avenues of future work ahead for AMPCS, building on its new architecture.

X. Acknowledgments

The author thanks the entire AMPCS team for their excellent, dedicated work on the Re-Architecture effort, and special thanks to Marti D. Verdugo for her technical leadership during the process. The author also thanks Deane Sibol, Eric D. Melin, and Mike Reid at The Johns Hopkins University Applied Physics Laboratory (APL) for offering their valuable knowledge and experience during the AMPCS CFDP requirements and design definition work, and also Robert E. Wiegand at GSFC for providing his help with the Java CFDP Software Library.

The work described in this paper was carried out at the Jet Propulsion Laboratory (JPL), California Institute of Technology (Caltech), under a contract with the National Aeronautics and Space Administration (NASA). The Multimission Ground System and Services Office (MGSS) sponsored both the AMPCS Re-Architecture and the AMPCS CFDP Development tasks.

XI. References

- [1] Fielding, R. T., “Architectural Styles and the Design of Network-based Software Architectures,” Ph.D. Dissertation, Information and Computer Science Dept., Univ. of California, Irvine, Irvine, CA, 2000.
- [2] Fielding, R., and Reschke, J., “Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content,” *IETF Documents* [online database], URL: <https://tools.ietf.org/html/rfc7231> [retrieved 20 April 2018].
- [3] Ray, T., “Using the CCSDS File Delivery Protocol (CFDP) on the Global Precipitation Measurement mission,” *International Telemetry Conference Proceedings*, Vol. 40, International Foundation for Telemetry, San Diego, CA, 2004.
- [4] The Consultative Committee for Space Data Systems, “Space Link Extension Services—Executive Summary,” CCSDS 910.0-G-2, 2006.