Kafka data streaming with the Tool Command Language (Tcl) - the “lazy bone” approach

Frank Morlang
German Aerospace Center (DLR)
frank.morlang@dlr.de

ABSTRACT

The paper describes Tcl usage in the context of realizing topic-consumer and topic-producer functionality without Kafka C language API "librdkafka" interfacing. An alternative benefiting from and based on Tcl’s process pipelining flexibility is presented. Its implementation in a cascaded multi-system data interrogation solution for holistic optimization purposes is shown and described.

Keywords
data streaming, process pipe, Apache Kafka

Article Received: 10 August 2020, Revised: 25 October 2020, Accepted: 18 November 2020

Introduction

Apache Kafka® **Error! Reference source not found.** is a widely used messaging and event streaming platform. Its advantages, especially if data river / lake implementations are concerned, can be condensed to the following aspects:

- **Distributed**

Kafka’s distributed architecture frees the potential of benefiting from partitioning and replication.

- **Throughput performance**

Dealing with high data amounts in combination with low latencies characterizes the fast handling capability, requested by today’s big data use cases.

- **Scalability**

Kafka is highly scalable. Scale of processing is enabled by the possibility of implementing consumer groups, combining the advantages of message queuing on the one hand and publish-subscribe on the other hand.

- **Fault tolerance**

Partition dispersion over Kafka cluster servers and responsibility sharing in terms of data handling by each server for partition sub-groups can easily be configured in a way, that partitions are duplicated over several servers, thus, generating a high degree of fault tolerance.
Against this background, Kafka was chosen as a data pipeline for the communication between different applications to establish a holistic optimization architecture (Figure 1) for collaborative trajectory, resources management and maintenance operations in aviation. Topics for the initialization of flight plans, their associated updates and the distribution of a master time represent the main communication streams. An initial flight plan generator and a master clock act as data producers, whereas an airport collaborative decision making (ACDM) simulation **Error! Reference source not found.** must deliver updates as well as react on ones generated by other applications. Therefore, both roles, producer and consumer needed an implementation.

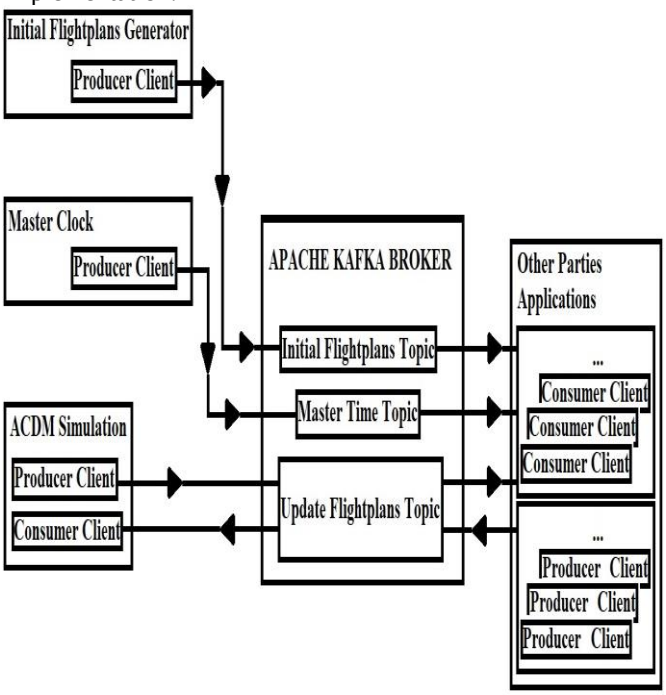

Figure 1. Holistic optimization architecture overview

Challenge

The initial flight plan generator and the master clock application (Figure 2) were developed in Tcl.

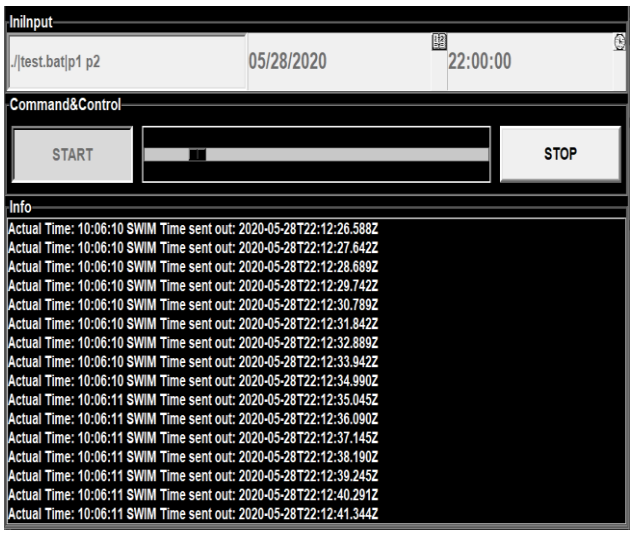

Figure 2. Master clock application

Consequently, it seemed appropriate to do the same for the Kafka connectivity. A search for usable Tcl packages revealed that KafkaTcl offers a Tcl interface to the Kafka C language API **Error! Reference source not found.**. It requires the Apache Kafka® protocol C library implementation "librdkafka" **Error! Reference source not found.** to be installed. Individual needs requested the realization on a computer running Windows 10. Microsoft Visual Studio 2015 build instructions are provided but building with GNU Compiler Collection (GCC) on Windows exposed problems, not short-time solvable because of environment specific constraints. The question for an alternative solution arose, either to develop a bridge application in a programming language with easier Kafka client establishment possibilities under the associated conditions, or to find another approach in Tcl.

Solution

The command line utilities the Kafka distribution comes with contain a producer as well as a consumer tool. Running these programs from Tcl with the standard input and output connected to file descriptors and communicate using pipes form an elegant and flexible solution. An example flight plan in extensible markup language (xml) format is shown in Figure 3. These flight plans are sent to a dedicated topic for initialization purposes. Other applications consuming this topic can setup themselves to a harmonized simulation startup condition, having all the same planned schedules.

```
<?xml version="1.0" encoding="UTF-8"?>
<Flight xsi:schemaLocation="http://www.w3.org/2001/XMLSchema ./flight.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.w3.org/2001/XMLSchema">
    <Station iataDesignator="FRA"/>
    <Addition/>
    <AdditionNext/>
    <AircraftType>E195</AircraftType>
    <AircraftTypeNext>E195</AircraftTypeNext>
    <Airline>OS</Airline>
    <AirlineNext>OS</AirlineNext>
    <Bound>1</Bound>
    <BoundNext>0</BoundNext>
    <OrgDest>SZG</OrgDest>
    <OrgDestNext>SZG</OrgDestNext>
    <Registration>OELWI</Registration>
    <RegistrationNext>OELWI</RegistrationNext>
    <ScheduledTimeIn>2019-11-26T06:10:00.000Z</ScheduledTimeIn>
    <ScheduledTimeOut>2019-11-26T07:00:00.000Z</ScheduledTimeOut>
    <Trip>261</Trip>
    <TripNext>262</TripNext>
    <Position>V178</Position>
    <PositionNext>V178</PositionNext>
    <Deicing>0</Deicing>
    <DeicingNext>0</DeicingNext>
</Flight>
```

Figure 3. Example flight plan

The code segment responsible for doing the job is shown in Figure 4. A writing command pipe is set to the console producer batch file, including the associated command line parameters:

- address and port of the target broker,
- path to the producer configuration file,
- name of the topic.

Each xml file is loaded to the document object model and gets a xml header appended. Before being sent out to the pipe, spaces, newlines and carriage returns are stripped away by regular expression substitutions as well as the encoding is set to utf-8.

```
set commandoPipe [open "|kafka-console-producer.bat \
        --broker-list $targetbroker:$targetport \
        --producer.config ../config/producer.properties \
        --topic iniflightplans" w]
...
..
.
foreach f $flightPlanFiles {
    set InDoc [dom parse [tDOM::xmlReadFile $f]]
    set Outputcontent ""
    append Outputcontent <?xml " " version="1.0" " " \
            encoding="utf-8"?> [$InDoc asXML]
    regsub -all {>\s+<} $Outputcontent {><} Outputcontent
    regsub -all {>\n<} $Outputcontent {><} Outputcontent
    regsub -all {>\r<} $Outputcontent {><} Outputcontent
    set Outputcontent [encoding convertto utf-8 $Outputcontent]
    puts $commandoPipe $Outputcontent
    flush $commandoPipe
    $InDoc delete
}
```

Figure 4. The “lazy bone” code segment (producer)

The code segment for reading from a topic is presented in Figure 5. A read channel is defined, configured and bound to a file event procedure in case of being readable by a reading command pipe, set to the console consumer batch file. Received content is processed in an own procedure.

```
proc processpipe {frompipe} {
    ...
    ..
    .
    return
}
proc readpipe {channel} {
    processpipe [read $channel]
    return
}
lassign [chan pipe] readChanId writeChanId
fconfigure $readChanId -blocking 0 -encoding utf-8
fileevent $readChanId readable [list readpipe $readChanId]
set commandoPipe [open "|kafka-console-consumer.bat \
        --bootstrap-server $targetbroker:$targetport \
        --consumer.config ../config/consumer.properties \
        --topic $targettopic" r]
```

Figure 5. The "lazy bone" code segment (consumer)

Results and discussion

Results of a solution test for sending 2880 initial flight plan xml files are shown in Figure 6 to Figure 8. It refers to a local test set-up (Table 1).

Computer	Lenovo MIIX 320-10ICR 4000 MB memory Intel® Atom(TM) x5-Z8350 CPU @ 1.44GHz Windows 10 Home 64bit (1909)
Tcl/Tk version	Tcl/Tk 8.6.10 (64 bit)

Table 1. Test set-up parameters

Deriving the piping performance (Figure 8) needs to consider that the throughput time (Figure 6) is mainly ruled by the xml loading and parsing duration (Figure 7). Regarding a piping performance of about 3 MB / sec (Figure 8) seems to be away from anything associated with "high performance". This impression gets its relativization by the following:

- Achieving high throughput in MB/sec is easier for large messages. For relatively small messages the domination of the overhead of processing each message must be taken into consideration.
- The test reflects a single producer thread set-up on a cheap computer (Table 1) representing low performance hardware.

For the given use case (Figure 1) the performance is more than sufficient (even with the low performance test hardware) enabling slim effort based Tcl/Tk developments doing a perfect job in providing connectivity in a complex multi-system holistic optimization approach with distributed Kafka data streaming.

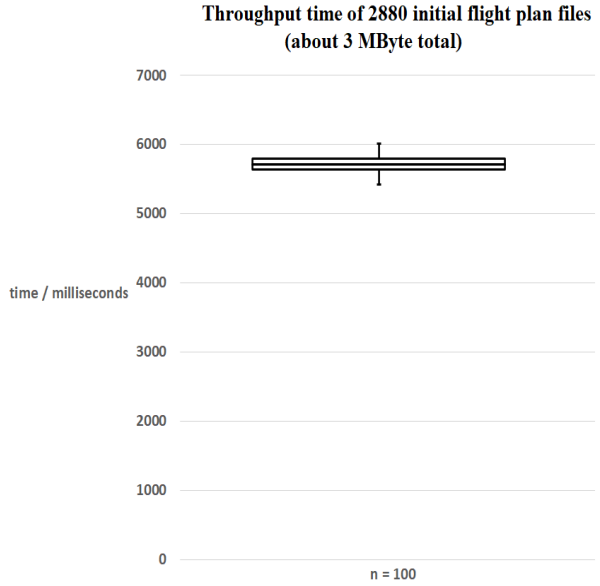

Figure 6. Throughput time

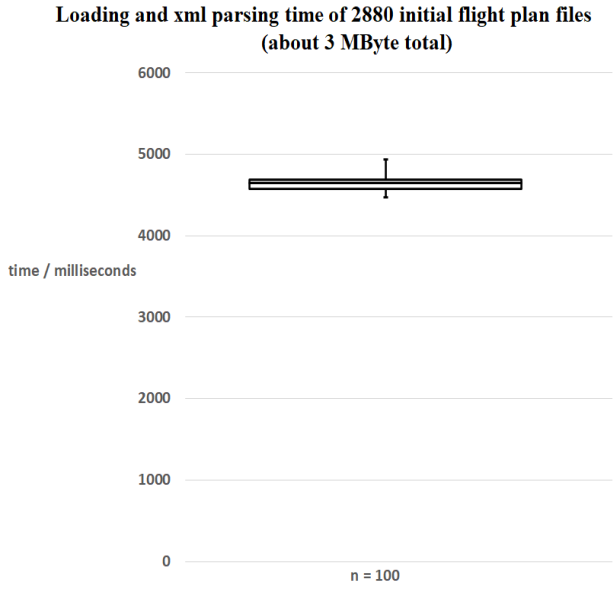

Figure 7. Loading and xml parsing time

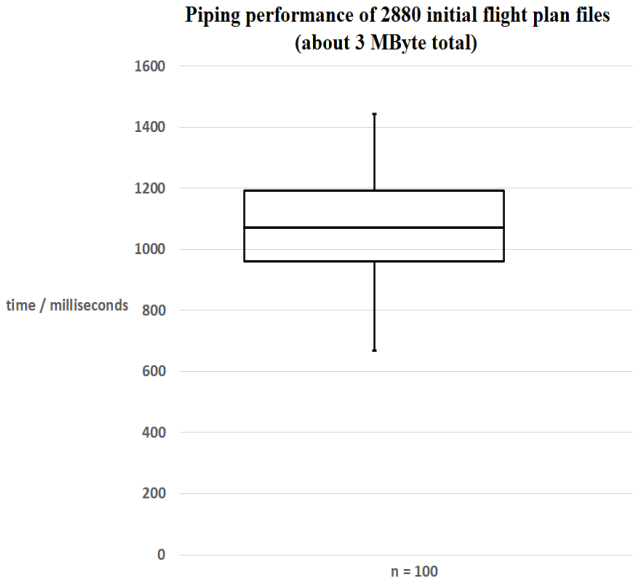

Figure 8. Piping performance

Outlook

Although the solution has a small source code footprint, actual considerations point in the direction of function segregation in an own Tcl package. Parallel investigations regard the potential of combining the source code of the Kafka C language API "librdkafka" **Error! Reference source not found.** with CriTcl **Error! Reference source not found.Error! Reference source not found.** for better performance by benefitting from C code runtime embedding. Future Kafka connectivity requests and their associated performance constraints will decide on the development intensity and direction.

Acknowledgment

This work has been conducted in the framework of the research project OPs-TIMAL, financed by the German Federal Ministry of Economic Affairs and Energy (BMWI).

References

[1] The Apache Software Foundation. “Apache kafka® - A distributed streaming platform”, [Online]. Available: https://kafka.apache.org/. [Accessed 28th Jul. 2020].

[2] Schier, S., Timmermann, F., Pett, T.: "AIRPORT MANAGEMENT IN THE BOX - A HUMAN-IN-THE-LOOP SIMULATION FOR ACDM AND AIRPORT MANAGEMENT", 65th German Aerospace Congress, Braunschweig, Germany, 2016

[3] flightaware / kafkatcl "Kafkatcl release v2.4.3", [Online]. Available: https://github.com/flightaware/kafkatcl/releases/tag/v2.4.3. [Accessed 20th Jul. 2020].

[4] Edenhill, M. "librdkafka - the Apache Kafka C/C++ client library", [Online]. Available: https://github.com/edenhill/librdkafka. [Accessed 20th Jul. 2020].

[5] Landers, S. & Wippler, J-C. (2002). CriTcl - Beyond Stubs and Compilers. In Proc. 9th Annual Tcl/Tk Conference, Tcl Community Association, Vancouver, Canada.

[6] Kupries, A. (2016). C Runtime in Tcl. In Proc. 23rd Annual Tcl/Tk Conference, Tcl Community Association, Texas, USA.

Author

Frank Morlang

1999 Diploma in Engineering in Materials Science of Technical University of Darmstadt

1999 - 2001 Project engineer at Aerodata company Braunschweig

2001 - today Researcher at German Aerospace Center (DLR)