Performance comparison of TwinCAT ADS for Python and Java*

Alexander Weber, Holger Eichelberger
{weber, eichelberger}@sse.uni-hildesheim.de
SSE, University of Hildesheim, Hildesheim

Svenja Wienrich, Per Schreiber
{reimer, schreiber}@ifw.uni-hannover.de
IFW, University of Hannover, Hannover

Abstract

Real-time and in-process measurements are important in the manufacturing domain, e.g., for real-time process monitoring. For performance reasons, such data is often processed in virtualized environments on edge devices, as e.g., provided by the company Beckhoff. For exploring modern AI methods, integration with high-level languages such as Python or even with Industry 4.0 platforms for advanced data flows is needed.

In this paper, we analyze the read/write performance of a Beckhoff device integrated via Python or Java. For our experiments, we use a simulation on a PC as well as a networked setup with a Beckhoff device. We show that the Java-based solution is faster than the Python one by 2-3 times. We also show that small arrays can be read as fast as a single value, that there is no difference between operations for small or big data types and that there is no difference between reading and writing data.

1 Introduction

In-process measurements are important for analyzing production problems in manufacturing and as a basis for adaptive production control using Artificial Intelligence (AI) methods. Production machines such as those for milling or welding offer high-speed real-time access to process data such as the spindle current or torque. Accessing these data streams, analyzing them with easy-to-develop AI methods in higher-level programming languages and controlling or optimizing the underlying machine operations are recent challenges.

A particular trend is to run AI methods on edge devices, e.g., on Phoenix Contact [3] or Beckhoff devices[1]. Moreover, collecting and processing data in a distributed fashion in an Industry 4.0 software platform is important in manufacturing. In the IIP-Ecosphere project[2] we work on a novel, AI-integrated platform [3] integrating Java and Python services. To allow for interoperability with Industry 4.0 devices, various connectors based on standardized or proprietary protocols must be provided by such a platform.

One frequently used protocol for real-time data access in factories is Beckhoff TwinCAT ADS (Automation Device Specification). TwinCAT is also the name of the programming environment for Beckhoff programmable logic controllers (PLC). To provide access to ADS data, we aim for a specialized platform connector. Thus, our research question is *which performance can be achieved for ADS real-time streams in high-level programming languages* such as Python or Java. We aim for a sampling rate of 1-20 kHz depending on the task at hands, for a platform integration the machine pace of 8 ms is sufficient [1].

In this paper, we analyze the read/write performance for different ADS data types and compare the performance of the Python `pyADS` library and our Java library. An integration of the native ADS library and the conversion of ADS data types to a higher-level language may cause overheads. We show that our Java integration based on JNA (Java Native Access)[3] is faster than an existing Python library, that read / write operations for all ADS data types operate per approach at a similar performance and that both approaches significantly drop in performance when utilised through a network.

Liang et al. built a system for a digital twin robot where ADS is one of the communication protocols for transferring data between the physical robot and its virtual counterpart [2]. The authors created a direct connection between the ROS (Robot Operating System) and the TwinCAT PLC and measured the average transmission speed for 16 data points at 9.45 ms through Ethernet. Further, Galeas et al. used in [4] ADS to control a telescope through an ADS-based Ethernet connection between a Linux and a TwinCAT PLC. The authors used parallel connections for writing and receiving data and achieved an average response time of 1 ms in an experiment with over 50,000 data points. However, no details on the implementation language were given. Both papers focus on the usage of ADS as a part of a system and evaluate its transmission times, but the authors do neither consider different types of ADS variables nor different implementation languages. In our work, we examine the performance of two different programming language integrations as well as access to ADS variables of different types.

*IIP-Ecosphere is partially supported by the German Federal Ministry of Economic Affairs and Climate Action (BMWK) under grants 01MK20006A and 01MK20006D.

[1] `https://www.beckhoff.com/de-de/`

[2] `https://www.iip-ecosphere.de/`

[3] `https://github.com/java-native-access/jna`

In the following, we firstly discuss in Section 2 our approaches to integrate ADS into Java and Python. In Section 3, we detail the setup of a performance experiment on accessing data with a simulated and a real Beckhoff device and evaluate the results. We end with a short conclusion and some directions for future work in Section 4.

2 Approach

For integrating a real-time data access approach into a software platform, it is important to understand tradeoffs between different forms of integration, in our case, in particular for Python and Java.

For Python we rely on the existing `pyADS` library[4]. However, for copyright reasons, `pyADS` does not ship with the native (Windows) TwinCAT ADS library, which can be obtained from an installation of the Beckhoff TwinCAT ADS development environment. For Java, so far no encompassing and comparable library exists. For an initial evaluation of performance tradeoffs, we implemented a simple Java library in the style of `pyADS` and execute native TwinCAT functions through JNA. JNA automagically binds a native library against the Java Native Interface (JNI) only based on Java interfaces of the native library. In contrast, JNI is known to operate at better performance while requiring low-level C programming.

Both libraries access ADS variables through their name and type as defined in the underlying PLC program. In more details, the access to an ADS variable consists of two library calls, namely `getIndexOffset` to resolve the variable name to a memory offset and the second (e.g., `AdsSyncReadWriteReqEx2`) to read / write data to a given memory offset. As ADS defines various data types ranging from Boolean, signed / unsigned Integers (`LWord`, `DWord`, `Word` or `SWord`, i.e., 64, 32, 16 and 8 bit Integers, respectively), Doubles, Strings, to composite types and fixed-length arrays of these types, both libraries provide specific read / write operations per type.

In contrast to Python, in Java a type and a value conversion is required. However, some ADS types exceed the bit length of the corresponding Java types and, thus, require a conversion to the next larger / smaller type when reading from or writing to ADS. For example, an unsigned ADS variable of `SInt` (8 bit) is stored in a signed Java `short` variable of 16 bit length. Values of the ADS types `Real` or `LReal` can directly be stored in the corresponding Java floating point data types `float` or `double`, respectively. However, for unsigned 64 bit ADS integers we have to resort to objects of the Java `BigInteger` class. When writing to ADS variables of unsigned types, we cast the bigger (signed) Java value down to the next fitting unsigned data type with the correct number of bits. For more complex data types, we employ a visitor pattern which linearly maps, e.g., an array or a composite type from / to memory through basic types operations at respective memory offsets. Moreover, we allow for caching already resolved ADS memory offsets.

[4] `https://pypi.org/project/pyads/`

3 Evaluation

We are particularly interested in the read and write performance (response times) from and to ADS using the approaches described in Section 2. The main goal is to determine whether the approaches can fulfill the aforementioned speed requirements of 1 kHz upwards (e.g., for sampling the 3 axis positions of a spindle) or at least the 8 ms machine pace. Further we are interested whether we can identify differences based on the used data type, especially given the required casts in Java, or whether arrays, e.g., for the 3 axis positions, have any advantage over accessing individual values. Lastly, differences may occur when we access ADS locally on the same device or via network.

The *experiment setup* involves a usual Laptop (Intel Core i7-8665U with 4 cores and 32GB of RAM) as well as a Beckhoff IPC C6930 (Intel Core i7-7700, 4 CPU cores with one core isolated for real-time tasks, 32 GB RAM, Windows 10 IoT Enterprise) PLC / edge device, both connected via a Gigabit managed Ethernet switch. As the PLC and the Beckhoff TwinCAT 3 programming environment is based on Windows as operating system, we also equipped the Laptop with Windows 10 and the original native TwinCAT `TcAdsDll` library version 2.11.0.41. The TwinCAT project for programming / simulating the PLC defines 14 variables of basic data types and 14 of corresponding array types, each array of length 3 (inspired by the three spindle axis case). Moreover, we are using Java 13+33 and Python (Cython) 3.8.10 on the Laptop, versions that are compliant with the requirements of the IIP-Ecosphere platform.

The *experimental subject* is a simple Java / Python program utilizing the respective approach from Section 2. Both programs access the same ADS variables in the same sequence, for experimental purposes in terms of measurement loops. As *experimental procedure*, we run 10 iterations of 1,500 access operations. A single iteration performs either read or write operations for one of the 14 base data types or their corresponding array type, respectively. We record the sum of the ADS calls per iteration allowing to easily discard potential warm-up iterations.

In a pre-experiment, we analyzed whether the JVM requires some warm up, e.g., for Just-in-Time compilation. Thereby, we did not find any indication for performance differences between the (initial) iterations allowing us to consider all collected data points.

Tables 1 and 2 illustrate selected results from the experiment[5]. In more details, we selected the results for the largest (64 bit) and the smallest (8 bit) data

[5] All experimental material is published on Zenodo (`https://zenodo.org/record/8421817`) and the Java ADS library as Open Source.

Table 1: Results for 1,500 local/simulated accesses [s]

	Python		Java	
ADS Data Type	***read***	***write***	***read***	***write***
LReal	6.116	5.734	1.949	1.923
ULInt	5.781	5.676	1.994	1.886
LInt	5.986	6.036	1.877	1.907
SInt	5.701	5.881	1.843	1.923
Byte	5.811	5.940	1.863	1.958
LReal Array	6.010	5.950	1.979	1.988
ULInt Array	5.775	5.952	2.103	1.931
LInt Array	6.281	5.797	2.009	2.023
SInt Array	5.780	6.034	2.047	1.964
Byte Array	5.795	5.953	2.010	2.080

Table 2: Results for 1,500 PLC accesses [s]

	Python		Java	
ADS Data Type	***read***	***write***	***read***	***write***
LReal	14.478	14.275	5.370	5.331
ULInt	14.447	14,572	5.329	5.486
LInt	14.471	14.598	5.412	5.415
SInt	14.357	14.470	5.434	5,384
Byte	14.486	14.451	5.436	5.484
LReal Array	14.437	14.600	5.507	5.420
ULInt Array	14.634	14.252	5.577	5.368
LInt Array	14.439	14.419	5.423	5.418
SInt Array	14.490	14.584	5.418	5.505
Byte Array	14.464	14,500	5.476	5.401

types. We do not display the complete data for all measured data types here as the left out data types have same access times, i.e., there seems to be no correlation between the data type and response times. When comparing the average time for read and write operations for all data types we found a small deviation 0.01% in either direction. In general, array accesses seem to be a bit slower than accesses for individual values, but only by 0.03%. Further, a comparison of the Java operations for `ULInt`, where we apply conversion to `BigInteger`, to the other 64 bit or even 8 bit data types, does not show any overhead. We also tracked memory usage and CPU usage during the experiment using PerfMon and, besides some singular spikes, did not identify relevant outliers or deviations.

Table 1 shows that accessing ADS values through Python on the same device is about three times slower than the Java approach. In other words, in Java a single read or write operation can be executed in 1.323 ms or 1.301 ms, respectively, while Python requires in average 3.9 ms for either operation. Thus, in a local setting, data access at the machine pace of 8 ms is possible for Java and Python. However, already a sampling rate of 1 kHz is problematic as at maximum the response times are up to 10% higher than the average while the average itself is 30% above our goal. If related values as spindle axes positions are stored in an array and accessed through a single operation, a (virtual) sampling rate of up to 3 kHz is possible.

Table 2 shows the results for network accesses to the Beckhoff PLC. Here, both approaches need between 2.5 and 3 times longer, while again there are no significant differences among reading or writing values of different data types. Also the machine pace of 8 ms is achievable, but for Python only by utilising arrays to read multiple data points at once.

Besides limitations of an initial experiment not paying attention to all disturbances on a Windows system, we did not perform experiments on Linux to use the original Beckhoff ADS library.

4 Conclusions and Future Work

Access to real-time information in production processes is important, but also performance critical. Due to our context, we performed an experiment with Java and Python for Beckhoff TwinCAT ADS. This experiment provides initial insights for higher-level programming languages and surprisingly shows that Java outperforms Python. In particular on Java, the requested 8 ms machine pace as well as sampling frequencies close to 1 kHz for individual values and slightly above 2 kHz for arrays of length 3 can be achieved. In the future, we plan to further investigate the performance differences between Java and Python, to optimize the Java implementation, e.g., to directly integrate the TwinCAT library through JNI or to apply the GraalVM[6], which compiles Java and Python into native code.

References

[1] H. Eichelberger, H. Stichweh, and C. Sauer. "Requirements for an AI-enabled Industry 4.0 Platform – Integrating Industrial and Scientific Views". In: *Intl. Conference on Advances and Trends in Software Engineering*. 2022, pp. 7–14.

[2] C.-J. Liang et al. "Real-time state synchronization between physical construction robots and process-level digital twins". In: *Construction Robotics* 6.1 (2022), pp. 57–73.

[3] H. Eichelberger, G. Palmer, and C. Niederee. "Developing an AI-enabled Industry 4.0 platform - Performance experiences on deploying AI onto an industrial edge device". In: *Softwaretechnik-Trends* 43.1 (Feb. 2023), pp. 35–37.

[4] P. Galeas et al. "EtherCAT as an alternative for the next generation real-time control system for telescopes". In: *Journal of Astronomical Telescopes, Instruments, and Systems* 9.1 (2023), pp. 017001–017001.

[6] `https://www.graalvm.org/`