ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP020

MANAGEMENT OF EPICS IOCs IN A DISTRIBUTED NETWORK ENVIRONMENT USING SALT

E. Blomley*, J. Gethmann, M. Schuh, A.-S. Müller
Karlsruhe Institute of Technology, Karlsruhe, Germany
S. Marsching, aquenos GmbH, Baden-Baden, Germany

Abstract

An EPICS-based control system typically consists of many individual IOCs, which can be distributed across many computers in a network. Managing hundreds of deployed IOCs, keeping track of where they are running, and providing operators with basic interaction capabilities can easily become a maintenance nightmare.

At the Institute for Beam Physics and Technology (IBPT) of the Karlsruhe Institute of Technology (KIT), we operate separate networks for our accelerators KARA and FLUTE and use the Salt Project to manage the IT infrastructure. Custom Salt states take care of deploying our IOCs across multiple servers directly from the code repositories, integrating them into the host operating system and monitoring infrastructure. In addition, this allows the integration into our GUI in order to enable operators to monitor and control the process for each IOC without requiring any specific knowledge of where and how that IOC is deployed. Therefore, we can maintain and scale to any number of IOCs on any numbers of hosts nearly effortlessly.

This paper presents the design of this system, discusses the tools and overall setup required to make it work, and shows off the integration into our GUI and monitoring systems.

ENVIRONMENT

The two accelerators KARA (Karlsruhe Research Accelerator) [1] and FLUTE (Far-infrared linac and test experiment) [2] each operate in separate, self-sufficient network environments, including DNS, DHCP, NTP and similar services. Most hosts are virtual machines running Ubuntu LTS versions for the operating system, managed via a Proxmox VE cluster [3]. Dedicated computer hardware is only in use if required. Examples are operator terminals in the control rooms, our data archiving cluster and time synchronization servers.

Most EPICS IOCs run on Ubuntu based virtual machines, as we try to avoid using dedicated hardware, such as VME crates. This requires most hardware being able to communicate via TCP or UDP, with serial communication being managed via serial-to-ethernet hardware gateways. The exception here are IOCs running directly on commercially available hardware, which in most cases is hardware for accelerator beam diagnostics, but recently also includes some power supplies.

For personal or machine protection-critical systems, PLCs are in use, which are also interfaced using EPICS IOCs via TCP. For data archiving of EPICS process variables (PVs) a Cassandra [4] NoSQL cluster is in use. The default graphical user interface is Control System Studio (CSS). Table 1 shows an overview over the numbers of server-side components.

Table 1: Control System Components in Numbers

	KARA	FLUTE
Virtual Machines	23	11
Physical Server	8	9
Registered IP Devices	707	341
PLCs	31	11

EPICS SOURCE CODE

Most EPICS related source code is managed via a site-specific GitLab instance [5]. While most of our EPICS related projects are not publicly visible at the moment, we are working on publishing more of the ones that are not closely tied to our infrastructure. Some custom EPICS modules are already available via our GitHub organisation *KIT-IBPT* [6].

EPICS Distribution

As we are almost exclusively running Ubuntu Linux inside our accelerator network server infrastructure, we build and distribute our own EPICS Debian Packages (.deb). We are running EPICS 7 and only compile, build and distribute EPICS modules which are used by at least one IOC. The most important modules are asyn, StreamDevice, autosave, motor, areaDetector, s7nodave, MRF, and Open62541. We make use of GitLab Continuous Integration (CI) for applying custom patches, building each component and distributing the resulting Debian packages via an internal APT repository. The EPICS version and the available modules are exactly the same across all EPICS servers.

IOC Structure

Typically, each IOC is represented by a repository in our GitLab instance. While we make use of the *makeBaseApp.pl* script, we also add an executable to the *iocBoot/iocLinux* folder. By default, the executable is just called *run*.

The IOCs are grouped into accelerator-specific GitLab groups. For devices which are used across both accelerators, such as certain power supplies, a group for shared IOCs exists. To simplify the IOC maintenance tasks in such cases, two different start-up files and two different executables *run_flute* and *run_kara* are stored, so that all other parts of the IOC code are shared. A similar approach is taken in

* edmund.blomely@kit.edu

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP020

case multiple instances of one IOC are required, with each instance getting a corresponding executable *run_**.

This allows starting the IOC directly after being cloned without any additional setup steps being required other than building the IOC using *make*. In addition, it allows defining certain run-time variables which can be used to simplify the start-up files in case of multi-instance or multi-accelerator IOCs. A *README.md* in the top directory typically covers the overall mission of the IOC as well as technical details regarding the record and start-up files, the relevant PVs and information regarding the hardware interface, if one exists.

Recently we started making use of Python softIOC [7]. These Python-based IOCs follow a completely different and less pre-defined structure, but we established the *run_ioc.py* as the default executable script to run the IOC similar to *iocBoot/iocLinux/run* for a regular C-based IOC.

Repository Features

With only a small number of people actively contributing to IOC development, no strict rules and workflows are established and it is left in the responsibility of the developer to make sure that any code change is correctly working. At the same time, especially for larger changes, using a Git branch and a merge request review workflow can be leveraged. The project specific issue tracker is used to track feature requests and bug reports. For the future, an IOC project specific automated CI pipeline is also foreseen to at least make sure that the IOC builds without errors and the executable *run* file(s) exists.

SALT INTEGRATION

To manage the IT infrastructure, the *Salt* project [8] has been used for some years now. Salt is a Python-based, open-source software for IT automation, remote task execution and configuration management following the *infrastructure as code* approach.

Salt State Files

The concept is based around the Salt master, a dedicated server storing the so-called *Salt state files* (SLS). Each state file can consist of multiple states, while each state describes some configuration which can be applied to a target. Examples for such states are the installation of OS software packages, managing configuration files for specific services or executing remote commands. But also more complex features, such as cloning code repositories, running web servers or deploying containers are supported. Overall around 620 individual states are defined, distributed across around 200 SLS files.

The syntax of the state files leverages YAML, making it human readable. In the *Top state file*, individual hosts or groups can be targeted, defining which states will be applied to which hosts. With the ability to include or require state files in other state files, flexible targeting mechanisms, and a templating engine for dynamic file handling, close to zero manual interaction or maintenance is required across all physical and virtual host systems. At the same time, scaling up the number of hosts for already existing groups can be done nearly effortlessly.

EPICS Server Configuration

An EPICS virtual machine server consists of around 500 individual states. Some of these states take care of the default configurations, such as account management, SSH keys, network mounts and software package roll out.

The EPICS specific states handle installation of our EPICS distribution, environment settings, PV bash completion scripts and a set of custom accelerator and site related Python modules [9].

IOC Configuration

The IOC configuration consists of one file for each accelerator representing the single-source-of-truth for the IOC deployment scheme.

Each entry in this file represents one IOC and must include a unique identifier for the IOC, the GitLab project name and the target EPICS server in form of the hostname. While these three parameters are enough for the full deployment, additional configuration parameters exist, which are either pre-defined with sensible default values or are only of informational character and therefore fully optional.

As most IOCs actually only run one instance, additional parameters need to be configured for multi-instance IOCs. Some parameters also dynamically change their default values based on other configuration parameters. For example, if the type parameter is set to *python*, the default *run-script* automatically changes to *run_ioc.py*. The group parameter behaves similarly. For the KARA IOC list, *kara* is the default group, but if *shared* is used, the *run-scripts* defaults to *iocBoot/iocLinux/run_kara*.

Table 2 shows the overview of the available IOC configuration parameters.

Table 2: Possible Parameters of the IOC Salt Configuration. *m*: Mandatory Parameter, *d*: Default Value for Most Cases, *o*: Optional

Parameter	Type	Description
IOC name	m	
Git name	m	Repository name
Host	m	Target server
Run script	d	Default: *run*
Group	d	FLUTE, KARA, shared
Type	d	*C* or Python
Auto start	d	Start on host reboot?
Critical	d	Trigger alarm if not running?
Dependencies	o	For non-default .debs
Git Branch	o	For test periods of features
Documentation	o	External documentation
Description	o	Short IOC description
Repository	o	For non-default repository

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP020

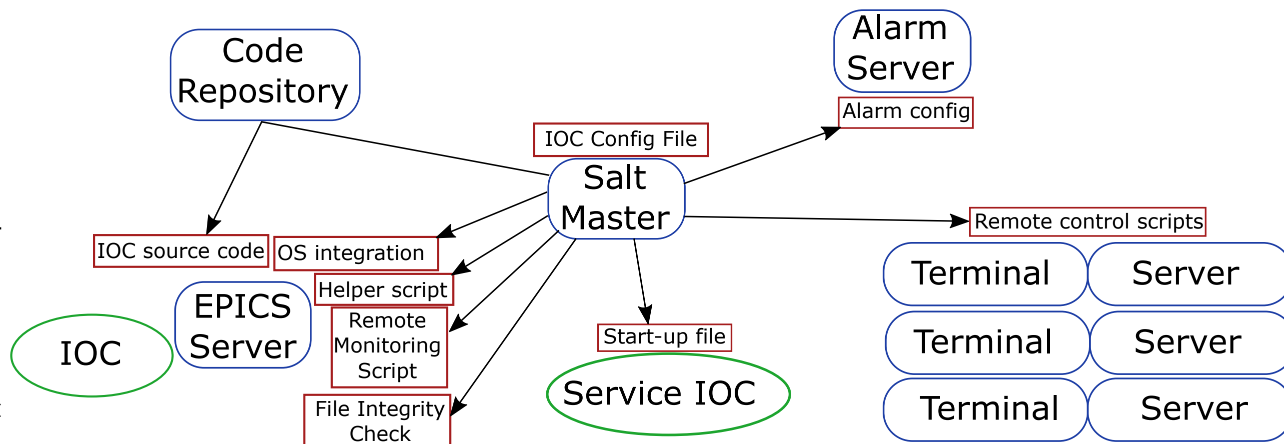

Figure 1: Workflow of the IOC deployment via Salt. Each arrow represents one component of the IOC environment discussed in more details in the text.

SERVER DEPLOYMENT

While all EPICS servers have exactly the same EPICS Base version, modules and extensions available, other reasons exist to distribute the IOCs on multiple servers. One reason is that direct communication with hardware, for example via USB, may be required. Another reason lies in the network layout, as only a specific host might have access to a special sub-network, acting as a gateway. One example are our digital cameras, which are all set up in a separate network, with only the dedicated camera EPICS server having access to this network. Table 3 shows an overview of the EPICS environment in numbers.

Table 3: EPICS Environment in Numbers

	KARA	FLUTE
EPICS VM Servers	4	2
EPICS Physical Servers	1	2
Deployed IOCs	94	42
Deployed Services	11	10
PVs	80,000	12,000

OS Integration

For each IOC, around nine Salt states are applied, taking care of the full deployment and integration of the IOC. While Salt is crucial in rolling out the IOCs, once deployed, the EPICS server and all IOCs are not relying on any external connection or online management link. Rolling out an IOC involves:

- Clone & build from code repository
- OS integration into *systemd*
- Create helper script
- Configure file integrity checks
- Setup remote monitoring script

Locally on the server, the IOC is registered in Ubuntu's service manager *systemd*. In addition, a helper script with a name, which is unique across all IOCs, is set up in the format *ioc-<unique-ioc-name>*. This system-agnostic script allows the basic interaction with the IOC, without requiring knowledge of any details of systemd and allows exchanging the actual OS integration transparently, if required in the future. Figure 1 shows the full deployment scheme.

Screen Session

The actual IOC runs inside a named screen session following the same naming scheme as the helper script. Connecting to the screen session is made possible with this helper script, allowing access to the IOC's terminal, if required.

File Integrity Checks

Typically, no manual adjustments to the deployed IOC files are needed. But in the process of investigating bugs, fixing issues or rolling out new features, it might happen that local modifications are applied to the IOC source and might be forgotten. Therefore, we also roll out local checks, which periodically scan all IOC directories. They look for local modifications, check that the Git branch is correct, compare the build time to the last start time of the IOC, and query the remote repository to see if the source code is up to date.

DISTRIBUTED IOC CONTROL

In addition to the helper script created for each IOC, we use Salt to create and distribute another helper script, which internally has the full list of IOC names and corresponding EPICS servers stored. This helper script is distributed across all EPICS servers and all operator terminals. It allows using commands such as *ioc-manage ioc-<unique-ioc-name> <command>* from any server or terminal to interact with the IOCs in the same way as if locally connected to the correct host. This script executes the given command via SSH, knowing on which server the IOC is running. To reduce latency and overhead for repeated usage of this script, SSH connection pooling is used.

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP020

IOC INTEGRATION

Leveraging the option to interact and check the status of all IOCs from any EPICS server, we set up a meta-IOC, which we named the *Service Manager IOC*. This IOC makes use of the *ioc-manage* script to query the status and allows starting and stopping each IOC.

A section of this IOC's start-up file is automatically generated via Salt, with one call to *dbLoadRecords()* for each configured IOC, providing all relevant information via macro substitutions. After a new start-up file has been pushed to the IOC, it is automatically restarted to pick up the changes immediately.

Execute Device Support

The integration of the *ioc-manage* script into the *Service Manager IOC* happens through the Execute device support [10], which was developed by aquenos GmbH on behalf of KIT.

This development was motivated by the observation that users often wanted to trigger actions outside the EPICS control system (e.g. send e-mails). Before this device support was available, this was often done through the *sub* or *aSub* record. Compared to the execute device support, this approach has two significant downsides: First, it requires at least some basic knowledge of the C programming language. Second, if not done properly, there is the risk of locking up the whole IOC due to performing asynchronous actions (like launching external programs) from the record's processing routine.

The Execute device support, in contrast, can easily be used by any user who if familiar with running programs from the shell. Data from EPICS can be passed through command-line arguments, environment variables, or on the standard input. Data from the executed program can be passed back to EPICS through the exit code or the standard output or standard error output.

Due to only relying on the operating system's infrastructure for executing programs, this approach is completely language-agnostic, so the user can use the programming language that they prefer and with which they are already familiar (e.g. shell scripts, Python, Perl, etc.). This is an advantage over language-specific solutions which run the user code inside the EPICS IOC like pyDevSup [11] or asyn [12], which both simplify the integration of custom code into EPICS IOCs but still require some knowledge about the internal structure of EPICS and are tied to the Python or C programming languages.

On a technical level, the device support provides output records for passing data to the execute program and input records for collecting data from the executed program, once it has finished execution. In addition to that, execution of the program can be triggered through a *bo* record.

The device support can operate in a mode where it waits for the external program to finish execution and subsequently triggers the processing of other records. This mode of operation is implemented trough asynchronous record processing, so that the IOC is not locked up while waiting for the program to finish execution.

The device support can also operate in a *fire and forget* mode, where it starts execution of the external program, but does not wait for this program to finish execution. In this mode, the program's exit status and output are not available, so this is typically used when an external action shall be triggered, but there is no need to pass any data back into EPICS (e.g. when sending notification e-mails).

GUI

For the operator of the accelerator the most interesting information is whether the IOC is actually running or not. We therefore have an overview panel, where all 100+ IOCs (at KARA) are displayed, including their status, buttons to stop and start the IOC, and direct links to the source code or documentation. In addition, the result of the *file integrity checks* is reported and graphical hints indicating whether an IOC is running on a development branch are shown, giving both the operators and IOC maintainers quick access to a full status overview of all IOCs. More information is displayed in each tooltip, for example on which host the IOC is running. Figure 2 shows an excerpt of the overview panel for KARA.

In a detailed panel for each IOC, all configuration and run-time parameters are provided (see Table 2). Figure 3 shows one example.

We currently still use Control System Studio (CSS), and therefore decided that adding new or removing obsolete IOCs from the GUI integration has to be done manually, which makes this the only manual step in the whole deployment. The automation effort on that level would be quite high compared to the benefit, especially in the context that IOCs are not added or removed on a daily basis and we are planning a switch to the successor of CSS anyway.

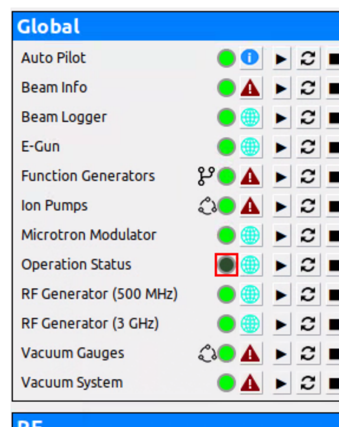

Figure 2: Part of IOC overview panel.

MONITORING

As with every component in controls environments, global monitoring and alarming should be considered.

Each IOC is therefore classified as either *critical* or *non-critical* in terms of importance to the operation of the accel-

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP020

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

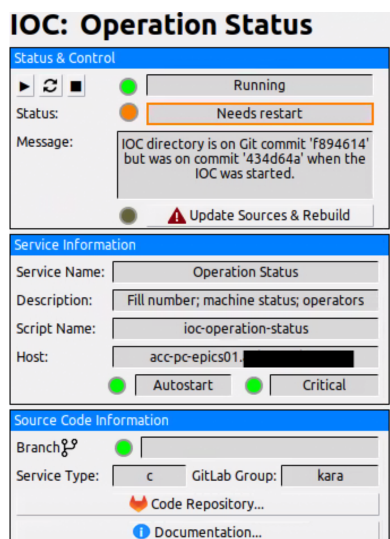

Figure 3: IOC Details panel.

erator. Examples for non-critical IOCs are the ones which are only used on demand, are still in development, testing or evaluation phase, or only provide auxiliary information.

EPICS Alarming

As this parameter is also part of the IOC configuration in Salt, we leverage this to dynamically define the severity field of the status PV to be either *MAJOR* or *NO_ALARM*. Our alarm server [13] will also get an updated alarm configuration via Salt and is restarted, if the service manager IOC start up file is changed and therefore picks up this change automatically. Due to how we set up our alarm tree and hierarchy, a critical IOC which is not running will immediately propagate to the top and will be very hard to miss for an operator sitting in the control room.

Remote Monitoring

As our accelerators are in operation regularly without operators being present in the control rooms, we also wanted to have a remote monitoring option and external notification. Our general IT monitoring tool, Checkmk [14], already looks into CPU, memory and hard-disk space usage as well as general health of IT hardware. It also provides the option to include simple local checks which either result in 0 or 1 when executed. It was therefore quite straightforward to generate an additional script per IOC via Salt during IOC deployment, providing this information to Checkmk.

The general features of such an IT monitoring tool perfectly supplement the local monitoring: e-mail notifications, if the IOC is in a critical state (not running) after a grace-period, long-term tracking of uptime, and also monitoring of the general health of the host system and thus indirectly of the CPU and memory usage of the IOCs.

SUMMARY & FUTURE PLANS

The overall setup of our IOC management evolved strongly over time. While running our IOCs in named *screen* sessions and using the local helper scripts was already established from the very beginning, it was still a manual job at that time. But only with the introduction of a tool such as Salt, it became feasible to increase the level of OS integration and add all the additional features summarized in this paper. Deployments of IOCs are consistent, it is very simple to add more IOCs and the status of each running IOC is easily available to anyone.

With the current iteration is also scalable to any level: It doesn't really matter if we manage 50, 100 or 1000s of IOCs in this way. At the same time, as neither the IOC nor EPICS need any modifications to run this system it can be and actually is also used for other services, which share similar characteristics to the IOC structure. Still, there are even more features planned:

IOC Creation

Currently, creating a new IOC is a manual process, although the actual steps involved are always the same. The idea here is to fully automate this process, which includes applying our few custom modifications to the basic IOC layout, selecting the desired EPICS modules, and creating the GitLab project, This way, consistent templates for IOC documentation and automatic testing can also be provided.

Continuous Integration

At the moment, the individual IOCs do not use CI features. We want to at least add some build checks and publication of the local IOC documentation.

Automated GUI Creation

Currently we have to create the CSS GUI integration manually. As we are planning to transition to the successor Phoebus [15] soon, we plan to re-evaluate the options to also fully automate the panel creation.

Embedded IOCs

We nowadays run well over 40+ IOCs on embedded hardware. Depending on the manufacturer, a certain level of external control and status checking of the embedded IOCs is possible, but this is not consistent. With some manufacturer-specific adjustments, we could very much apply a similar level of management using the same approach and integration as with our own IOCs and services.

Containerized IOCs

While to a certain degree our layout can be seen as a light version of a containerized IOC setup, there of course are additional benefits to using a true container, such as IOC specific EPICS environments. Changing to a full container workflow can easily be done by only adjusting the OS integration level. At the same time, we do not see a dramatic benefit of doing so, as the dynamic scalability of processing capabilities and hosts is not required and our setup actually is rather static. But it still being considered as a potential option for the future.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP020

REFERENCES

[1] Karlsruhe Research Accelerator (KARA), https://ibpt.kit.edu/kara

[2] Far-infrared linac and test experiment (FLUTE), https://ibpt.kit.edu/flute

[3] Proxmox Virtual Environment, https://www.proxmox.com/en/proxmox-virtual-environment

[4] S. Marsching, "Scalable Archiving with the Cassandra Archiver for CSS", in *Proc. ICALEPCS'13*, San Francisco, CA, USA, Oct. 2013, paper TUPPC004, pp. 554–557.

[5] KIT-IBPT GitLab, https://gitlab.kit.edu/kit/ibpt/

[6] KIT-IBPT GitHub, https://github.com/orgs/KIT-IBPT/

[7] Python softIOC, https://github.com/dls-controls/pythonSoftIOC

[8] Salt Project, https://saltproject.io/

[9] J. Gethmann, E. Blomley, S. Marsching, W. Mexner, A.-S. Müller, P. Schreiber, and *et al.*, "Simple Python Interface to Facility-Specific Infrastructure", in *Proc. 13th Int. Workshop Emerging Technol. Sci. Facil. Controls (PCaPAC'22)*, Dolní Brežany, Czech Republic, Oct. 2022, pp. 51–53. doi:10.18429/JACoW-PCaPAC2022-THPP9

[10] EPICS Execute Device Support, https://github.com/KIT-IBPT/epics-execute

[11] EPICS pyDevSup module, https://mdavidsaver.github.io/pyDevSup

[12] EPICS asyn module, https://epics-modules.github.io/asyn

[13] N. J. Smale *et al.*, "The ANKA Control System: On a Path to the Future", in *Proc. ICALEPCS'13*, San Francisco, CA, USA, Oct. 2013, paper MOPPC099, pp. 336–339.

[14] IT Monitoring Platform checkmk, https://checkmk.com

[15] CSS Phoebus, https://control-system-studio.readthedocs.io

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI