Erkios: End-to-End Field-Based RAS Testing

Luis E. Oleas Chávez, David E. Bakken, Anjan Bose
School of Electrical Engineering and Computer Science
Washington State University
Pullman, Washington, USA
loleasch@eecs.wsu.edu, bakken@wsu.edu, bose@wsu.edu

Patrick Panciatici
Power System Expertise Department
RTE France
Versailles Cedex, France
patrick.panciatici@rte-france.com

***Abstract*—Today's increasingly-stressed grids are relying more on remedial actions schemes (RAS) as an emergency tripwire to stop a blackout. One problem associated with RAS is that once they are deployed only some of their components are tested, and then infrequently. In this paper we introduce Erkios, a middleware framework for testing RAS end-to-end and in-field. This paper presents an analysis of the requirements for such testing, the design of Erkios including its failures handled, and a description of the Erkios prototype.**

***Index Terms*—fault tolerant systems, middleware, power system protection, remedial action scheme, substation protection.**

I. Introduction

Today's power grids are increasingly stressed due to a myriad of factors including insufficient transmission growth and integration of renewables. As a result, remedial action schemes (RAS) [1] are increasingly deployed. Such schemes have extremely high requirements for the performance and availability of the communications system and the distributed components that it interconnects: sensors, actuators, and RAS logic in the control center or in a substation [2][3].

RAS schemes are tested when installed but then go many years without being tested. This is problematic. The issue is: if they are needed 5–10 years later, will they really work then?

To address this, we have developed *Erkios*[1], a middleware framework for testing RAS schemes (and, in the future), other such mission critical functionalities. Erkios is designed so that the power applications and sensors can be unaware of its presence; it is "off to the side" of the operational RAS system except for a few simple interception points.

Erkios must *ipso facto* employ a sophisticated and integrated suite of fault tolerance mechanisms. This is because, by design, it must detect, and when possible, compensate for a wide range of failures (and combinations thereof). This includes component failures of both the power grid and of Erkios itself. The techniques utilized include heartbeats, timeouts, and self-testing mechanisms to compensate for the slight change in the data value inherent in converting from digital to analog (to be inserted into a sensor instead of the PT/CT) and then back to digital.

We have drawn from many sources to design Erkios, including a wide range of RAS schemes. We have also drawn from the fields of distributed and dependable computing as well as application domains including spacecraft, microprocessors, and wireless sensor networks.

We know of no system that does what Erkios does while being "off to the side", i.e., letting the RAS applications, sensors, actuators, control center software, and other items be unaware of it (configuring Erkios does not require these components to be aware of it). There is one related system that is similar, except for requiring application-awareness – i.e., the applications have to be written to use it in order to reap its benefits. This is Adaptive Reconfigurable Mobile Objects of Reliability (ARMOR) middleware [4]. ARMOR implements multi-threaded processes to manage redundant resources across interconnected nodes, detect errors in user applications and infrastructural components; and provide failure recovery. ARMOR has also not been evaluated for suitability for testing RAS schemes other than in the thesis of the first author [5].

The contributions of this paper include the following (the detailed requirements such as **R7** are given later):

- Analysis of requirements for in-field, end-to-end testing of RAS and other real-time centralized defense systems leading to **R1**, **R2**, **R3** and **R6**.
- Design of a middleware framework implementing the above contributions and requirements, namely in-field, self-testing for a wide range of RAS.
- A prototype implementation demonstrating that a system that meets Requirements **R1 –R7** can be built.

The remainder of this paper is organized as follows. Sec. II overviews RAS while Sec. III presents Erkios' design requirements. Sec. IV describes Erkios' architecture. Sec V

[1]Erkios is Greek for "Defender of the House", a title given to Zeus.

This research was funded in part by RTE France Contract # 4500534551, US DOE Award Number DE-OE0000097 (TCIPG), and a scholarship from the government of Ecuador through the SENESCYT program.

explains the failure model assumed for Erkios and the grid components involved. Sec. VI describes language and configuration details of Erkios. Sec. VII concludes.

II. Remedial Action Schemes

Remedial Action Schemes (RAS) are sometimes known as System Integrity Protection Schemes (SIPS) and, in the past, as Special Protection Schemes (SPS) [1]. These schemes go past what traditional protection and control schemes can do to avoid known contingencies and failure modes with pre-planned actions specific to each. RAS go beyond this to protect the grid against much more rare system emergencies, typically with sensors from more locations than a typical protection scheme. They are designed to "minimize the potential and extent of widespread outages that could result from more serious but less common or anticipated events" [1].

RAS can have logic at a control center, a substation, or both. For example, control center logic can arm the RAS logic in the substations its actuators are in. In other cases, all of the RAS logic may be in the control center, as is the case with Southern California Edison's C-RAS configuration supporting 14 RAS. In the other extreme, all of the logic may be distributed in substations with no logic at all in a control center.

Fig. 1 depicts the canonical structure for a RAS. It accommodates the above range of RAS configurations plus all with sufficient details that we could find in the literature.

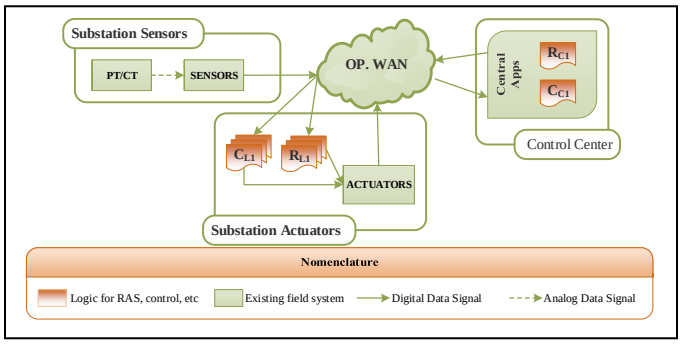

Figure 1: Canonical RAS Structure

Sensors in substations consist of the Potential Transformer/Current Transformer (PT/CT), which provide sensing capabilities feeding analog signals into the sensor proper. This provides analog-to-digital conversion and outputs data in a standardized format such as the IEEE C37.118 standard for synchrophasor data.

Sensor data is delivered by the operational wide-area network (Op. WAN) to other substations and the control center, which may contain logic for RAS and control regimes.

Actuators are in some substations (shown as mutually exclusive in Fig. 1 but there can be overlap in practice). These devices receive commands to open or close from RAS logic that may be in the same substation or remotely from the control center.

III. Erkios Design Requirements

Erkios was designed to meet the requirements that are summarized as follows (for full details and detailed justification see [5]):

R1. The system shall be able to remotely disable actuators before the system verifies the correct operation of RAS, as well as re-enable them once the test has been completed.

R2. The system shall be able to remotely inject a false analog signal into a sensor, instead of the normal PT/CT inputs.

R3. The system shall perform in-field tests according to the kind of RAS and Self-Testing coding technique selected by the user.

R4. The system shall be able to log the tests performed to check test data flow through the End-to-End system and its result.

R5. The system shall detect, log and report to the user any failures existing in the operational system.

R6. The system must not require application programs to be aware of its presence, because it will be used to test many existing systems that cannot be modified.

R7. The system shall be able to inject a given data value, have it traverse multiple modules, in which data can be converted from digital-analog then analog-digital, and detect if the same value, or one sufficiently close was delivered at the end.

These cover the configurations from Sec. II as well as including as many RAS papers and other documentation as we could find; for example, [5] has a 25-page appendix with detailed use cases drawn from these.

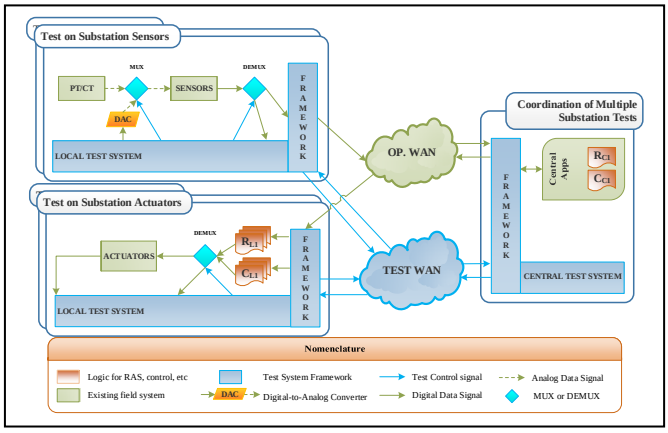

Figure 2: Erkios End-to-End, In-Field Architecture

IV. Architecture of Erkios

We have extended the canonical RAS architecture in Fig. 1 in order to implement the requirements given in Sec. III. This is depicted in Fig. 2. In this figure, the existing items in the field are in the same colors as in Fig. 1, namely operational components are in green and RAS or control logic is in

orange. In order to meet the requirements, we have added, inline in the flow of sensor data and actuator commands, multiplexors (MUX) and demultiplexors (DEMUX). These are depicted in turquoise. Finally, the test system components are depicted in blue. We note that a test component labelled "FRAMEWORK" can be thought of a pass-through element that has the option of logging what goes through it. We now describe these elements in turn.

The substation sensors have their PT/CT generate an analog signal that is shown sent to the sensor in Fig. 1. However, in order to be able to insert arbitrary values into the sensor, we have inserted a MUX between the PT/CT and the sensor. This is remotely controllable by the Erkios. When the MUX is configured to do so, it takes its input not from the PT/CT but rather the local test system (LTS) in the substation, via a digital-to-analog converter (DAC). After the sensor a DEMUX can be controlled to send the output sensor data to the OpWAN, the LTS, or both.

The control center of course houses the traditional control applications that can include RAS and control. It also includes a central test system (CTS) that orchestrates the actions of multiple LTSs in order to coordinate an end-to-end test involving multiple substations.

Actuator substations are augmented with a DEMUX after the RAS logic (which can be thought of as an empty pass-through if the logic is all centralized) but before the actuator. This enables Erkios to test the RAS from sensor to RAS logic and ensure that the correct command to the actuator is sent in the right circumstances yet reroute it to prevent the RAS from taking the drastic action it is commanded to do.

V. Erkios Failure Model

In this section we describe Erkios' failure model. First, we give motivation on how remote client-server interactions are much more complicated than local object calls to the same process on the same computer. We also define terms necessary to understand Erkios' compound failure model. These two subsections are intended as background for readers from a power background. Next, we explain the failures that Erkios covers. Finally, we discuss how and where it handles them.

A. Distributed Computing and Failure Concepts

The client-server paradigm is a common building block in distributed computing [6]. Here, a client sends a request message to a server. The server then performs some operation on behalf of the client.

A foundational problem in distributed computing is that, if no reply comes back, the client cannot (without external help) tell if the request or reply message was dropped or if the server failed before, during, or after processing the client's request [7]. It is for this kind of reason that Erkios must employ a complex and well-considered suite of fault-tolerance mechanisms.

B. Failure Concepts

We now define basic terminology from the field of dependable computing [8]. These have been developed since the 1980s to provide a rich description of the issues and mechanisms involved with dependable computing, and in particular how they relate to each other [9] (a foundational book that has been translated into English, French, German, Italian, and Japanese).

A *fault* is the hypothesized or adjudged event that begins a component's problems. This can include such things as a lightning strike or a dormant bug in the logic of hardware or software. If not treated, a fault can lead to an *error*, which is an incorrect internal state. Examples here include a flipped bit in a register and a wrong byte on a disk. If errors are not handled they can become a *failure*, when a component does not meet its specification. Note that failures are observable outside of the given component in contrast to errors. Further, if Component A depends on Component B, then the failure of B is a fault in A when B's failure is observable (when A interacts with B).

Failures come under the category of *omissive* and *assertive* failures. The combination of these two categories is *arbitrary* failures.

Omissive failures involve a component not performing an interaction as specified. A *crash* failure is where a component halts permanently yet "cleanly", i.e. without making any externally-visible mistakes. *Omission* failures are where a component periodically omits a specified interaction (e.g., sending a message). A *timing* failure is when a component is late (or early) in performing a specified interaction (which of course must include a temporal characterization).

Assertive failures are when interactions are not performed to specification. A *syntactic* failure is when the interaction has the wrong structure; e.g., sending a floating-point number instead of an integer or having a malformed data structure. A *semantic* failure is an interaction with an incorrect meaning such as a temperature sensor below absolute zero or otherwise incorrect in the semantics of the application. Semantic failures are sometimes called *value* failures.

Arbitrary failures are the union of omissive failures, which occur in the time domain, and assertive failures, which occur in the value domain. Further, arbitrary failures can involve an adversary with complete knowledge of the system, including its configuration and algorithms, and that can cause the details, timing, and location of the failures to be at the absolute worst time for a given algorithm or for the system as a whole. *Byzantine* failures are a subset of arbitrary that involve sending bad values, lying about one's identity, and sending inconsistent semantic faults ("two-faced" behavior where a component tells one component that received one value and tells another component that it received different value).

C. Erkios Failure Model

Table I depicts the failure model that Erkios supports. Here, the *Deployment Category* indicates if the device exists in the *Field* in today's grid, was *Added Inline* to support Erkios, or is a purely "off to the side" Erkios component. Further, the failures are denoted by the first letter of the failure; e.g., "C" for crash failures.

TABLE I. ERKIOS FAILURE MODEL

#	Comp. Type	Area Test/Failure	Deployment Category	Failures Handled				
				C	O	T	V	B
1	L	PT/CT	Field	N	N	N	N	N
2	M	Pre-Sensor Switch	Added Inline	Y	Y	Y	Y	N
3	L	Sensor	Field	Y	Y	Y	Y	N
4	D	Post-Sensor Switch	Added Inline	Y	Y	Y	N	N
5	L	Op. WAN	Field	Y	Y	Y	Y	N
6a	R	Central RAS and Control Logic	Field	Y	Y	Y	Y	N
6b	R	Local RAS and Control Logic	Field	Y	Y	Y	Y	N
7	D	Actuator Switch	Added Inline	Y	Y	Y	N	N
8	L	Actuator	Field	Y	Y	Y	Y	N
9	E	CTS	Off to side ERKIOS	N	N	N	N	N
10	E	Test WAN	Off to side ERKIOS	N	N	N	N	N
11	E	LTS	Off to side ERKIOS	N	N	N	N	N

COMPONENTS

L LEGACY SYSTEM R RAS/CONTROL LOGIC M MUX D DEMUX E ERKIOS

D. *Erkios Failure Handling*

Fig. 3 depicts where the different failures are handled in Erkios. We now overview how and where failure handling is done; for more details see [5].

Erkios assumes that communication between the CTS and other components is always successful. This is a common assumption in helping to simplify distributed algorithms and is built with a thin layer above Java RMI that retransmits until the message gets through and also filters duplicates.

Erkios employs timeouts, heartbeat messages, and self-testing in similar ways for most components and in ways that are commonplace in dependable computing infrastructures. For example, periodic *heartbeat* messages are sent out by the CTS, LTS, and Test WAN. These modules also employ timeout mechanisms to translate crash and omission failures into timing ones by using timeouts. The combination of hearbeats and timeouts enable the LTS and CTS to determine with much more accuracy, which components have failed in what way.

The MUX and DEMUX modules in this software-only version also employ timeouts and heartbeats as above. However, in the field these techniques may not be available on their hardware equivalents so end-to-end testing is used to help infer their failure with greater accuracy.

Self-testing techniques are also necessary. A command from the LTS is converted to analog by a DAC then sampled and converted back to digital by the sensor. Due to inherent limits in precision, the original data word may not be bit-for-bit identical to what is received after both conversions. Thus, Erkios supports three different self-testing techniques — Berger Codes, Duplication Code (Swap & Compare), and Checksum Codes (Residue) — to provide different ways with which to detect that the digital value sent from the sensor is very close to the digital value sent to the DAC and then to the sensor as an analog signal.

VI. ERKIOS SOFTWARE

Erkios was written in Java, which can call to other programming languages when used with CORBA. Its test harness includes a data emulator in order to inject failures in various ways to support the different detection techniques. We summarize the software and its use here; for details see [5].

Erkios has a GUI that depicts the progress of a test in a manner identical to Fig. 2. It also allows for configuring

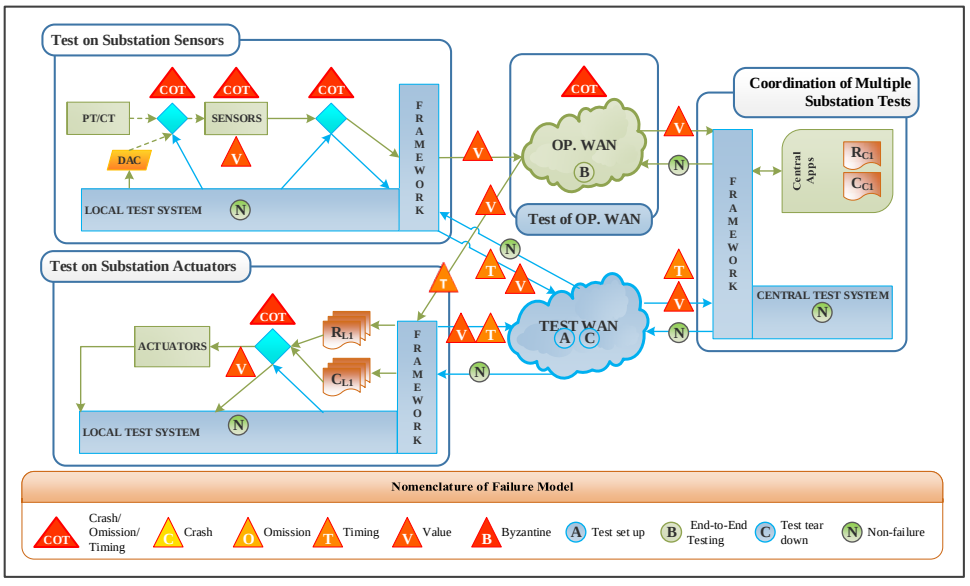

Figure 3: Erikios Failure Handling

different facets of a test. These include drop-down menus for RAS Classification, Type of RAS Scheme, Type of Control Signal, Component Test, Self-Testing. It also includes boxes to input values for different sensor variables used in the RAS. It also includes a scroll-down log window with tabs for log readings for Remedial Action, Trip command, ST Code, ST Code Contingency, List of Contingencies, RAS Test, Status Components, Links, and Contingencies.

The CTS orchestrates the components involved in an experiment by installing a Java implementation of each use case as selected by the configuration menu options. The values from the RAS variables are passed as parameters. A database is used by the CTS and LTS to log progress of all steps.

The architecture of Erkios and its use has been demonstrated with this software implementation as a first step. It has not yet been integrated with actual substation hardware but of course has been carefully designed for this.

VII. Discussion

A. Conclusions

In this paper we have introduced Erkios, a middleware framework to do end-to-end, in-field testing of RAS schemes. Erkios was designed to test a very wide range of RAS schemes in the field in order to ensure that the distributed components in a RAS scheme are functioning property. Erkios supports a very wide range of failure models handling failures of both grid components as well as pieces of Erkios itself.

B. Future Work

In the future we plan on integrating Erkios with actual sensor, actuator, and RAS controller hardware. This will involve interfacing with the data delivery portions of these devices in the manner depicted in Fig. 2. Such interfacing will include the ability of Erkios to send IEC 61850 GOOSE messages to devices and receive replies from them via GOOSE. Also valuable will be to extend Erkios so that it can invoke the self-testing modes of such devices and inspect the results, whether by GOOSE or other means. These mechanisms will complement, not replace, the testing described in this paper.

Future work will also include utilizing a combination of Erkios' off-to-the-side approach with ARMOR's application-aware requirements. The combination of these two approaches has the potential to cover a broader range of RAS schemes with better failure coverage than either approach can in practice do alone.

We also plan on extending Erkios in a number of ways to make it more resilient. For example, a pre-sensor MUX can have a timeout to revert to the normal, non-testing case of the PT/CT input being passed on to the sensor in the event that a command to stop the test (switch back to passing through the PT/CT signal) is not received after a certain amount of time. This would enable the system to revert to a safe state in the event that the Test WAN failed.

Adding access control and other cyber security features are obvious extensions, but ones we felt were premature until we had fleshed out and understood a full Erkios prototype. We also plan on extending the failure model to handle the Byzantine failure of many of the components, both existing field ones, components added inline, and off-to-the-side Erkios components.

Erkios at present only conducts one test at a time with settings chosen by hand with a GUI as described in Sec. VI. Adding a layer above it that allows for campaigns of experiments to be run whereby for each experiment one can specify the test inputs to different devices, timing issues, and the expected results (including timing). We also hope to extend this by integration in the DETER testbed and GridStat [2],[10] in order to do systematic emulation of WAN conditions at a large scale. By using a controllable WAN environment such as DETER, campaigns of experiments may be deployed under a wide range of WAN conditions. This will allow for the identification of RAS failure modes with respect to a multidimensional spectra of WAN operating points including those induced by rare events which are by definition difficult to observe in a real WAN. Work is already underway at WSU that integrates the primary USC DETER cluster with a testbed laboratory at WSU which contains real world RAS equipment and real-time simulation technologies.

Acknowledgment

We thank Shwetha Niddodi, Ryan Goodfellow, Anurag Srivastava, Carl Hauser, Thibault Prevoset, and Alenandre Parisot for their helpful feedback on this research.

References

[1] S. Horowitz, D. Novosel, V. Madani, and M. Adamiak, "System-Wide Protection", IEEE Power & Energy Magazine, 6(5), pp. 34–42, September 2008.

[2] D. Bakken, A. Bose, C. Hauser, D. Whitehead, and G. Zweigle, "Smart Generation and Transmission with Coherent, Real-Time Data", *Proc. IEEE*, 99(6), pp. 928–951, June 2011.

[3] G. Zweigle, "Emerging Wide-Area Power Applications with Mission Critical Data Delivery Requirements", in *Smart Grids: Clouds, Communications, Open Source, and Automation*, D. Bakken and K. Iniewski, Ed. Boca Raton: CRC Press, 2014, pp. X–Y, ISBN 9781482206111.

[4] Z. Kalbarczyk, R. Iyer, L. Wang, "Application Fault Tolerance with ARMOR middleware", *IEEE Internet Computing*, vol. 2, pp. 28–37, Mar. 2005.

[5] L. Chávez, "Erkios: End-to-End Field-Based RAS Testing for Enhanced Blackout Protection, M.S. Thesis, Dept. of Elec. Eng. And Comp. Sci., Washington State U., August 2014.

[6] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design, 5ed.* Boston: Addison-Wesley, 2011.

[7] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with One Family Faulty Process", *Journal of the ACM*, 32(2), pp. 374–382, April 1985.

[8] [ALR+04] A. Avižienis. J.C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing", IEEE Trans. Dependable and Secure Computing, 1(1), pp. 11–33, January 2004.

[9] [Lap92] J.C. Laprie, ed., Dependability: Basic Concepts and Terminology. New York: Springer-Verlag, 1992.

[10] [GBB+13] R. Goodfellow. R. Braden, T. Benzel, and D. Bakken, "First Steps Towards Scientific Cyber-Security Experiments in Wide-Area Cyber-Physical Systems", in *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop*, ACM, Oak Ridge, TN, Jan. 2013.