Global and local implications in programming languages

Pierre Depaz

Paris-3 Sorbonne Nouvelle

Undone Computer Science 07.02.23

Introduction

This contribution examines the notion of scaling through programming languages.

What I will present this morning is a contribution on the relationship between local and global in computer science, and particularly in software of engineering, and how it supports the design and implementation of particular pieces of software in the real world.

And in order to examine this relationship of local and global, we will examine the notions through the prism of geography.

Introduction

One of the goals of software engineering is the ability to *scale*.

Tsing, A. L. (2012). On Nonscalability: The Living World Is Not Amenable to Precision-Nested Scales. Common Knowledge, 18(3), 505–524.

This was mentioned yesterday by Florence Maraninchi, as she talked about expensibility and generality

Introduction

Scaling **up**: adding more resources to a machine

Scaling **out**: adding more machines with the same resources

Scaling **down**: considering smaller software teams

Boddie, J. (2000). Do we ever really scale down? IEEE Software, 17(5), 79, 81.

Laitinen, M. (2000). Scaling down is hard to do. IEEE Software, 17(5), 78, 80.

Martin Kleppmann. (2017). Designing Data Intensive Applications The Big Ideas Behind Reliable, Scalable, And Maintainable Systems.

Northrop, L., Feiler, P., Gabriel, R. P., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D., Sullivan, K., & Wallnau, K. (2006). Ultra-Large-Scale Systems: The Software Challenge of the Future (ADA610356). Software Engineering Institute.

The impetus to scale positively (scaling up, or scaling out) is something that seems to be taken for granted in software engineering research, a research that is focused on dealing with more data more efficiently, rather than consider whether one could deal with less data.

Introduction

By 2040, the IT sector is estimated to account for 14% of world emissions.

1/3 of these emissions come from software.

Lees Perasso E., Vateau C., Domon F., avec les contributions de Aiouch Y., Chanoine A., Corbet L., Drapeau P., Ollion L., Vigneron V. Prunel D., Ouffoué G., Mahasenga R., Orgelet J., Bordage F. et Esquerre P. (2022). *Evaluation environnementale des équipements et infrastructures numériques en France. Etats des Lieux et Pistes d'Action.*

Podder, S., Burden, A., Singh, S. K., & Maruca, R. (2020, September 18). How Green Is Your Software? Harvard Business Review.

The context that prompts such a study is the increasing part that the IT sector is taking in carbon emissions and climate change. That is, emissions are increasing, even though they should be decreasing.

These emissions are roughly split between 78% of hardware production of terminals, and 21% of (software) use of these terminals.

Introduction

Research on programming languages and sustainable development is largely focused on optimizing energy usage.

Manotas, I., Bird, C., Zhang, R., Shepherd, D., Jaspan, C., Sadowski, C., Pollock, L., & Clause, J. (2016). An empirical study of practitioners' perspectives on green software engineering. Proceedings of the 38th International Conference on Software Engineering.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J., & Saraiva, J. (2017). Energy efficiency across programming languages: How do energy, time, and memory relate?

Huber, S., Lorey, T., & Felderer, M. (2023). Techniques for Improving the Energy Efficiency of Mobile Apps: A Taxonomy and Systematic Literature Review. 2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA).

Kp, G., Pierre, G., & Rouvoy, R. (2023). Studying the Energy Consumption of Stream Processing Engines in the Cloud. 2023 IEEE International Conference on Cloud Engineering (IC2E).

During the past decade, a lot of the research on software development and carbon emissions has been focused on optimizing the footprint of a programming language in terms of speed and energy consumption, within particular computing platforms: mobile applications, or cloud applications.

It's important to develop a way to optimize, not just for speed and efficiency, but also for energy. New metrics also help to frame things differently.

Introduction

The Jevons paradox observes that increased efficiency is linked to increased consumption.

Saunders, H. D., & Tsao, J. Y. (2012). Rebound effects for lighting. Energy Policy, 49, 477–478. https://doi.org/10.1016/j.enpol.2012.06.050

Wirth, N. (1995). A plea for lean software. Computer, 28(2), 64–68.

York, R., & McGee, J. A. (2016). Understanding the Jevons paradox. Environmental Sociology, 2(1), 77–87.

However, optimization might not be the exclusive solution to the problem of energy waste and abuse.

The Jevons paradox phrases it nicely, named by a British economist who was puzzled by the fact that the steam engine, which was an optimization of energy usage of coal, was accompanied by an increase in coal consumption.

Jeffrey Tsao and his team have found similar results in the context of LED technology.

Similarly for “lean software”, as Niklaus Wirth was observing that improvements in hardware were compensated by bloat in software.

Introduction

Scénario	S1 Générations Frugales	S2 Coopérations Territoriales	S3 Technologies Vertes	S4 Pari Réparateur
Technique	Règne du low-tech	Numérique au service du développement territorial	Technologies pour décarboner	Innovations tous azimuts

ADEME (2021), Prospective - Transitions 2050 - Résumé exécutif, Horizons.

Crozat, S. (2023). Low-technicisation et numérique [Séminaire de recherche]. Seminar on Digital Environmental Policies, Centre Internet et Société.

In order to mitigate the dramatic effects of climate change, there are multiple possibilities. The french agency for ecological transition, ADEME, lays out 4 different scenarios to reach the emissions goal by 2050 (Ademe, 2021). But often engineering schools focus on scenarios 3 and 4, bypassing scenarios 1 and 2.

The current goal of the system I would like to question here is the necessity for a software system to *scale*. Not questioning the goal to scale means that we remain within scenarios 3 and 4 of the Ademe (where more technology is the solution), and ignore the scenarios 1 and 2 (where more technology is the problem).

Introduction

PLACES TO INTERVENE IN A SYSTEM

(in increasing order of effectiveness)

9. Constants, parameters, numbers (subsidies, taxes, standards).

8. Regulating negative feedback loops.

7. Driving positive feedback loops.

6. Material flows and nodes of material intersection.

5. Information flows.

4. The rules of the system (incentives, punishments, constraints).

3. The distribution of power over the rules of the system.

2. The goals of the system.

1. The mindset or paradigm out of which the system — its goals, power structure, rules, its culture — arises.

Meadows, D. (1999). Leverage Points: Places to Intervene in a System. The Sustainability Institute.

This research can also be inscribed in the framework of Donella Meadows' Places to Intervene in a System. As one of the co-authors of the Limits to Growth report, she highlights the different ways one can adjust a system to meet varisou goals.

It has different points, with different amounts of granularity. Here we would suggest to focus less on points 9-6, and rather 4-2, and particularly the goals of the system.

Implementation details over the whole lifecycle.

Software = design + programming + use

Simon, T., Rust, P., Rouvoy, R., & Penhoat, J. (2023, June 5). Uncovering the Environmental Impact of Software Life Cycle. International Conference on Information and Communications Technology for Sustainability. https://inria.hal.science/hal-04082263

So, in order to complement research on the strictly technical energy efficiency of programming languages, it's also necessary to study the implications of programming languages within the higher-level leverage points, but also the design and the use of programming languages.

Global, local and scale as geographical concepts.

Herod, A. (2010). Scale. Routledge.

Hess, D. J. (2009). Localist Movements in a Global Economy: Sustainability, Justice, and Urban Development in the United States. MIT Press.

But in order to understand scale as more than “doing more”, the assumption here is that we can benefit from interdisciplinarity as a shift in perspective.

If we start by looking at it naïvely, there seems to be an overall negative connotation of globality, following a period of hype (e.g. McLuhan's global village, or the global computing of the late 1990s-early 2000s). For instance, the process of bringing economic and political activities up to a global level, e.g. the phenomenon of globalization, has had some detractors. Localism, thinking locally, etc. have gained some traction in recent years (locally-sourced products, local communities, etc.).

So this seems, at first, to be a somewhat simple dichotomy of global things being bad, and local things being good.

Scale as (1) ideal concepts or (2) socio-material productions.

Hart, J. F. (1982). The Highest Form Of The Geographer's Art*. Annals of the Association of American Geographers, 72(1), 1–29.

Smith, N. (1992). Geography, Difference and the Politics of Scale. In J. Doherty, E. Graham, & M. Malek (Eds.), Postmodernism and the Social Sciences. Palgrave Macmillan UK.

In geography, the state of the art in this field is essentially divided between an idealist conception and a materialist conception.

For those who idealists, the local and the global are seen as part of a pre-existing conceptual matrix of scales within which social life is lived. As such, they are simply cognitive devices for enveloping and ordering processes and practices so that these may be distinguished and separated from one another as part of a hierarchy of resolutions – a particular process or set of social practices can thus be considered to be 'local' whereas others are considered to be 'global' in scope. Within such an analytical framework, the 'global' is usually defined by the geologically given limits of the earth, whereas the 'local' is seen as a spatial resolution useful for comprehending processes and practices which occur at geographical ranges smaller than the 'regional' scale, and where proximity makes sense.

For materialists, on the other hand, the key aspect of geographical scale is to understand that scales are socially and materially produced through processes of struggle and compromise. Hence, for instance, the 'national' scale is not simply a scale which exists in a logical hierarchy between the global and the regional but, instead, is a scale that had to be actively created through economic, technological and political processes, such as signing treaties, minting coins, and building roads.

Introduction

The position we take here is that programming languages take part in constitute global or local scales.

They are *intentionally* produced artefacts with a *purpose.*

Leonardi, P.M. (2010). Digital materiality? How artifacts without matter, matter. First Monday, Volume 15, Number 6 - 7 June 2010.

Kitchin, R., & Dodge, M. (2011). Code/Space: Software and Everyday Life. MIT Press.

Turner, R. (2014). Programming Languages as Technical Artifacts. Philosophy & Technology, 27(3), 377–397.

This morning, we intend to show how programming languages can be considered part of the processes constituting these scales.

First because programming languages create software, and because software norms our interactions with space, by deploying apparatuses for synchronicity and immediacy.

Second, because programming languages are the digital material of software. That is, they make ideas become manifest, they allow the embodiment of a model within practical limits (implementation). Finally, as technical artefacts, they have an intent and a function, and that function can related to processes of scaling.

The way we're going to work through this is therefore through the lens of software studies, which considers the digital objects such as programming languages are actants in a network, to take Latour's terminology, and that they contribute to the construction of the world..

Introduction

How do programming languages contribute to understanding of scales?

How can they help us think about *scaling down*?

Introduction

1. GoogleSQL and globality
2. Automerge and local-first
3. PeachCloud and material cloud

We will proceed with three different case studies, highlighting the features and contexts of three different ways in which programming languages are involvedin the production of scale.

First, we will be looking at this impetus for globalization in software and how it is decoupled from materiality, via the example GoogleSQL.

Second, we will inquire into the relationship between being local, and being offline, and whether this has a strict 1:1 equivalence, highlighting concepts of dependence with JavaScript and the automerge library.

In the third part, we will consider locality from a material perspective with the role of Rust in the PeachCloud project.

In conclusion, we'll suggest some further avenues for crossovers between social sciences and computer science, via the reframing of certain terms.

1. Constituting globality

1. Constituting globality

Programming languages abstract the locality of the machine.

Ritchie, D. M. (1993). The Development of the C Language. Second History of Programming Language Conference.

Wirth, N. (2003). The Essence of Programming Languages. In L. Böszörményi & P. Schojer (Eds.), Modular Programming Languages (pp. 1–11). Springer. https://doi.org/10.1007/978-3-540-45213-3_1

If we consider the local as the physically immediate, programming languages have historically tended to hide this physicality.

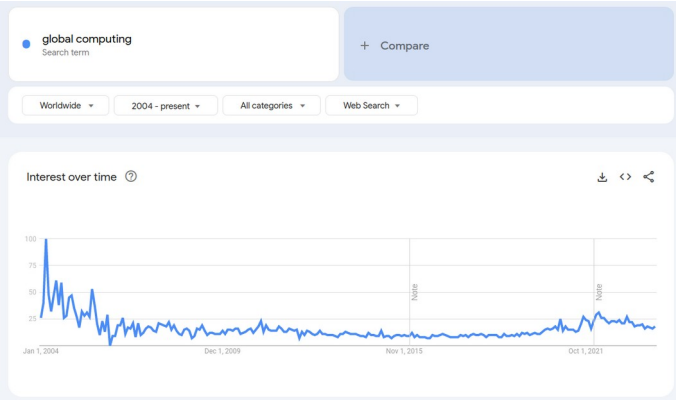
The advent of portable operating systems, epitomized with UNIX in the late 1970s, also suggested a move to _portability_: the ability for a programming language to support translation of a software onto another platform. This is particularly salient in the case of the C programming language, whose portability both supported the broadcasting of the UNIX operating system and was accelerated by the diffusion of UNIX. As such, the operating system enabled a layer of abstraction and becomes the new local, erasing the hardware.

As we follow this dynamic of higher level, higher abstraction, and higher portablity, throughout operating systems, virtual machines and sandboxed environments, one path leads us to WebAssembly, a language re-creating a virtual machine in the web browser. WebAssembly, by returing to the roots of Assembler programming, resets a cycle of development, by re-creating a starting point for any software to be run anywhere, a recursive loop that also highlights the relativity of the distinction between local and global.

This first case of globalization is thus a dynamic of abstraction of the individual machine.

1. Constituting globality

Global computing as predecessor of cloud computing, through networking



Google. (2024). Google Trends. Google Trends. https://trends.google.com/trends/explore?date=all&q=global%20computing&hl=en

Undone Computer Science 07.02.23

A second aspect of the portability of software comes with its distribution infrastructure. The World Wide Web enabled the wide sharing of software, by passing traditional supply chains. With the design and adoption of the Common Gateway Interface, it's not longer clear which machine is doing the computation.

Indeed, the term global computing is trending somewhat concommitantly with the advent of the commercialized Web.

1. Constituting globality

Multiple networked machines produce information, which can be managed.

One of the answers to this problem is BigQuery.

Melnik, S., Gubarev, A., Long, J. J., Romer, G., Shivakumar, S., Tolton, M., & Vassilakis, T. (2010). Dremel: Interactive analysis of web-scale datasets. Proc. of the 36th Int'l Conf on Very Large Data Bases, 330–339. http://www.vldb2010.org/accept.htm

Along with this dynamic of decoupling software from hardware, and further obfuscating the material presence of the machine, another dynamic is that of large-scale data creation. Programming is an information management technique which, in turn, produces new information, and this information soon exceeds the processing capacity of a single local machine.

In this case, programming languages constitute part of a solution to deal with large scale, distributed data. One of the most popular, and most large-scale solutions to this problem is BigQuery. BigQuery is Google's data warehouse, launched in 2010, and it provides a web interface to a broader query system called Dremel.

1. Constituting globality

GoogleSQL is the query language from Google's BigQuery data exploration system:

- query language
- procedural language
- data definition
- data manipulation
- data control

Google. (2024, January 31). Introduction to SQL in BigQuery | Google Cloud. Google Cloud. https://cloud.google.com/bigquery/docs/introduction-sql

Undone Computer Science 07.02.23

In order to interact with those systems, you need to use GoogleSQL, a multi-faceted *ad hoc* programming language, which allows the user to productively harness the larger and larger amounts of data that are made available.

It's an SQL dialect, but also has procedural aspects. The procedural aspects are ways to do control flow within this environment, and therefore to treat SQL queries as statement within statements, thus making the large-scale of data creation more easily manipulable.

GoogleSQL, can be considered here as one of the participants in the mediation of globalized exploitation of corporate data unifying a network of localized hardware through a software layer.

1. Constituting globality

Language with an ecosystem:
- data warehouses
- Google Cloud Platform
- financial limits

Franklin, M., Halevy, A., & Maier, D. (2005). From databases to dataspaces: A new abstraction for information management. ACM SIGMOD Record, 34(4)

GoogleSQL also does not exist in a void, but in a network of dependencies and constraints. The Google Cloud Platform creates dependence on a cloud corporation.

This implies a kind of pay-as-you-go programming, highlighting the financial limits, but not the energy limits, of what you can do with the programming language.

Concurrently, we can also identify a discursive reconceptualization of data storage and management systems: from databases to dataspaces, a dynamic that follows this process of abstraction and co-existence of various, disjointed pieces of data operating under different varieties of schemas and out of the control of the user.

1. Constituting globality

MapReduce as a programming paradigm to further abstract the machines.

Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. Communications of the ACM, 51(1).

GoogleSQL also exists in the vicinity of other programming techniques to handle this global scale, specifically to deal with that distribution of multiple groups of machines across geographic localities.

This movement can be seen in particular in the MapReduce programming model, which was first described in a paper by Google employees (Dean and Ghemawat, 2005). _MapReduce_ is a programming model (not quite a programming language) which essentially aims at unifying diverse sources of data. Originally designed for the Google web search service, global service if there is one, the way that _MapReduce_ proposes to operate is through a first step of harmonization (mapping the diverse/unprocessed inputs to harmonized/normalized outputs), followed by a second step of conflation, in which diverse inputs are operated on in order to collapse into a single output, effectively merging (or abstracting) the multiple into one.

Interestingly, MapReduce paradigm only proves to be more effective when multiple machines are involved to complete the computation. If only one machine is needed as a data source, it has no particular benefits.

1. Constituting globality

```
// Tuning parameters: use at most 2000
// machines and 100 MB of memory per task
spec.set_machines[2000];
spec.set_map_megabytes[100];
spec.set_reduce_megabytes[100];

// Now run it
MapReduceResult result;
if [!MapReduce[spec, &result]] abort[];

// Done: 'result' structure contains info
// about counters, time taken, number of
// machines used, etc.
```


Indeed, we can even see the mention of machines as ontologically grouped in the source code of the original paper:

The modularity and “summoning” of machines is embedded in the MapReduce proposed paradigm.

1. Constituting globality

Infrastructure as code further automates the hardware.

So we have this requirement to deal with multiple machines, across different geographical locations.

But global infrastructure is still infrastructure, and as such there is a need to materialize the physical things. This requirement manifests itself through programming languages, under a paradigm of configuration languages, or _infrastructure as code_.

There are generally two approaches to IaC: declarative (functional) vs. imperative (procedural). The difference between the declarative and the imperative approach is essentially 'what' versus 'how' . The declarative approach focuses on what the eventual target configuration should be; the imperative focuses on how the infrastructure is to be changed to meet this.

Is it then that it's better to use imperative over declarative? To know how things get done rather than automagically getting there? This I am unsure about, but the fact is that we work with new languages to work with new scales.

1. Constituting globality

Configuration languages to summon machines:

- Chef
- Puppet
- Terraform

```
resource "aws_instance" "server" {
  count = 4 # create four similar EC2
instances

  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"

  tags = {
    Name = "Server ${count.index}"
  }
}
```

HashiCorp. (2023). The count Meta-Argument—Configuration Language. Terraform | HashiCorp Developer. https://developer.hashicorp.com/terraform/language/meta-arguments/count

Undone Computer Science 07.02.23

It is no longer about running single applications on a local computer, but rather swaths of individual machines connecting to a single piece of software.

Infrastructure as code is that infrastructure is actually subdued as code. It is integrated into the software itself how many machines should be spinned up or taken down.

Code is expendable, people don't really count their CPU cycles, so infrastructure as code might be enabling a waste of resources and obliviousness of what the physical implications are. Making infrastructure as code is the epitomy of "scaling up".

1. Constituting globality

The reification of globalisation takes place through making the increasing abstraction of increasing amounts of materials usable.

So in this case we can see how programming languages take an active part in making the results of globalization actionable (that is, enabling activities across the globe and untethered to the specifics of a geographical or material connection).

They do this through the general process of abstraction, and by developing new programming languages to harness the new problems that emerge with scale.

As such, the feature of programming languages to create abstractions can be mapped to the abstracting processes of global business intelligence.

2. Imaginations of the local

We now turn to how programming languages can contribute to a conception of the local.

2. Imaginations of the local

Localhost and offline as the local, a space that is physically and intuitively graspable.

Standing as a negative definition of "global computing", "distributed computing", or “cloud computing”, "local computing" is first and foremost related to a specific machine. Furthermore, we can look at it not just as a specific machine, but also through its status with regards to network interfacing, as illustrated with the term `localhost`, a hostname which loops back to itself instead of coming from another node on a network.

In this case we're not so much talking about a physical device as a networked device: there is a different conception of locality which emerges with the popularization of the web. The individual processor architecture is abstracted away, but the differentiation becomes access to the network.

2. Imaginations of the local

Default assumptions of being online to interact with software.

(Moodle, Google Docs, etherpad, MS Python, ...)

Gwynne, S., & Kuligowski, E. (2010). The faults with default. Proceedings of the 2010 Interflam Conference. Interflam, London, UK.

Leyva, J. (2016, October 19). Access Your Learning Offline. Moodle.

JohnMcLear. (2013). Don't create network traffic unless needed · Issue #1384· ether/etherpad-lite. GitHub.

A few cases can highlight this "_online-first_" imperative, in which the networked is considered to be the default way of being in the (computing) world.

As the covid-induced distance learning has shown, not everyone has always access to a reliable network, a problem which was previously identified but understudied. In this case, locality, considered as exclusion from globality (through distance learning), becomes very apparent.

In the case of Moodle, there was a gap of three years between the release of their mobile application and the feature update which allowed users to work offline.

In the case of online document editing, such as Google Docs or Etherpad, the intersection of their growing ubiquity and network requirements highlights another default scale at which the network operates: one cannot edit, or see a document if one is not online.

Finally, the case of the recent release of Python in the Microsoft Excel spreadsheet highlights the role of dependencies. Python is a language which runs perfectly fine on Windows and MacOS machines, but the Excel implementation requires an active network connection as the Python code is being interpreted on Microsoft's servers, one of the reasons being to automatically manage Python modules as dependencies.

2. Imaginations of the local

Network connectivity as a **space of dependence**.

A space within which the value circulating is essential to one's needs.

Cox, K. R. (1998). Spaces of dependence, spaces of engagement and the politics of scale, or: Looking for local politics. Political Geography, 17(1)

One geographical definition of the local vs. the global is the identification of, and agreement with, the thing which one depends on (Cox, 1998).

Kevin Cox formulates the local as contingent with a space of dependence: the space within which the value circulating is essential to one's needs.

In our previous examples, the space of dependence is therefore network connectivity broadly understood, and the large amount of polluting cloud infrastructure that comes with it.

Deciding to manage somewhat manually one's own dependencies might therefore be seen as an act of constructing a local-first computing environment, rather than a global-first.

2. Imaginations of the local

Automerge is a software library implemented as a Conflict-Free Replicated Data Type (CRDT).

Shapiro, M., Preguiça, N., Baquero, C., & Zawirski, M. (2011). Conflict-Free Replicated Data Types. In X. Défago, F. Petit, & V. Villain (Eds.), Stabilization, Safety, and Security of Distributed Systems (pp. 386–400). Springer

Kleppmann, M., & Beresford, A. R. (2018). Automerge: Real-time data sync between edge devices. Self-published.

Jahns, K. (2021, June 11). How we made Jupyter Notebooks collaborative with Yjs. Medium.

So one way to redefine this space of dependence is to bypass this network requirement. One of the attempts to do so can be seen in the automerge data structure, which enables seamless offline and online collaboration.

CRDTs are fairly recent in their development, but did not start with Automerge. In fact, they were already proposed by a french computer scientist, Marc Shapiro, and his colleague, Nuno Preguiça. Proposed in multiple scientific articles in the mid-2000s, the concurrent advent of Google Docs in 2006 made the technology immediately obsolete, according to Shapiro.

CRDTs have nonetheless been gaining traction recently, as the implication of operating within certain spaces of dependencies becomes more obvious. For instance, Jupyter Notebooks, a popular data science app, restored its collaboration tools using CRDTs after Google got rid of the cloud service it had previously depended on.

2. Imaginations of the local

An original JS implementation, proof of application in prototypal software (e.g. **PushPin**, **Peritext**).

Rewrite in Rust for production.

Litt, G., Lim, S., Kleppmann, M., & van Hardenberg, P. (2022). Peritext: A CRDT for Collaborative Rich Text Editing. Proceedings of the ACM on Human-Computer Interaction, 6(CSCW2), 531:1-531:36.

van Hardenberg, P. (2023, January 17). Automerge 2.0 | Automerge CRDT. Automerge.Org. https://automerge.github.io/blog/automerge-2/

CRDTs are data structures that allow each user to edit their local copy of a document, and which ensure that different users' copies can be cleanly merged into a consistent result. Such data structures enable two clients to edit the same document, without consequences in terms of network connectivity: the canonical state of the document is resolved without conflict when a client regains network access and receives updates from other clients. In that sense, CRDTs change the online dependency in the software.

The original implementation was written in JavaScript, and the rationale for that was to have a flexible language to experiment with the data type at a theoretical level, before re-writing it in Rust for a production-ready environment. In this case, using choosing JavaScript allows for a flexibility of experimentation.

2. Imaginations of the local

Discursive manifestation with “local-first”

“The key difference between traditional systems and local-first systems is not an absence of servers, but a change in their responsibilities: they are in a supporting role, not the source of truth.”

Kleppmann, M., Wiggins, A., van Hardenberg, P., & McGranaghan, M. (2019). Local-first software: You own your data, in spite of the cloud. Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.

Through a very concrete technical research into a data structure, there are actually two things:

A manifesto for an ideal of scale (a so-called `local-first` approach) which facilitates the framing of the scale at which software is supposed to operate

Anda also a concrete, material contribution to making operating at this scale a realistic possibility.

2. Imaginations of the local

Languages and data structures: ideas through agnosticity, *de*-scaling as a **space for engagement.**

So what Automerge is an example of is the gradual movement from idea, to broad JavaScript implementation, to careful Rust rewrite, showcasing the way in which ideas are gradually reified into efficient software systems.

It is also an example as introducing a (somewhat) new data structure for a new conception of the scale(s) at which one works, rather than optimizing an existing one.

Nonetheless, it's important to point out that Ink&Switch never mentions energy efficiency, and this is my personal framing: by reducing a reliance on the cloud, one also reduces the footprint of the cloud. What they do advocate for is a new way of working, and thus a different way to engage with the use of technology.

Finally, the use of a certain programming language is only one part of the contribution, especially since JS is not very energy efficient, but it allows you to test an idea easily. It's also about devising a discursive framing with manifestoes, something which Cox calls spaces of engagement, networks of involvement that go beyond a strict data structure and its implementation.

3. Material and social engagements

We now turn to our last example, focusing on Rust and materiality.

3. Material and social engagements

Scales can be seen the result of social actors engaging in discursive organization.

Lempert, M., & Summerson Carr, E. (Eds.). (2016). Scale: Discourse and Dimensions of Social Life. University of California Press.

As discussed by Lempert and Summerson Carr, the scales that social actors rely upon to organize, interpret, orient, and act in their worlds are not given but made, sometimes rather laboriously so, traditionally by using natural languages and in our case, also by using programming languages.

This example focuses on the interaction of scale, sociality and technology through the material instantiation of a social network based on the SecureScuttleButt protocol.

3. Material and social engagements

SecureScuttleButt (SSB) is a protocol for gossip-like, machine-mediated, social interactions.

Tarr, D., Lavoie, E., Meyer, A., & Tschudin, C. (2019). Secure Scuttlebutt: An Identity-Centric Protocol for Subjective and Decentralized Applications. Proceedings of the 6th ACM Conference on Information-Centric Networking.

SecureScuttleButt's protocol design is centered on user-first, and LAN first.

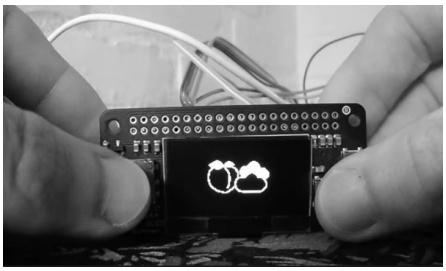
It has its roots in the gossip protocol family. These are based on models of rumor distribution, or epidemics - essentially a more or less random selection of peers to whom the information will spread. The approach is therefore clear: the aim is to define a technical protocol as a simulation of natural phenomena, based on the principle that information will have social patterns within it.

SSB's starting axiom is also offline availability: the protocol and its applications must be usable when not connected to the Internet. All data storage and access takes place locally, and updating or synchronization of this data takes place when, and if, there is a connection to another member of the network. SSB replicates, in a protocol-based way, the experience of life at sea - a local experience if ever there was one - by allowing synchronization between two peers only if they are connected to the same WiFi network.

3. Material and social engagements

PeachCloud is the physically concrete version of SSB

"*We aim to return (cloud) computing back into our homes and local communities in a way which fosters increased trust in one another and the socio-technical systems we inhabit*"



PeachCloud. (2019, November 9). Introduction. PeachCloud: Developer Documentation. https://mixmix.github.io/peach-devdocs/chapter_1.html

However, nothing is ever completely global, or completely local, and therefore SSB also includes a way for peers to centralize information, via special client peers called Pubs.

While a pub could just be run as another daemon on one's machine, the PeachCloud project was an initiative to bundle the SSB software necessary to run a pub into an easy-to-use, all-in-one hardware solution.

Using Rust as a need to engage materially.

A systems programming language feels more welcoming than Python.

As such, PeachCloud is an alternative to the cloud, using Rust as an alternative embedded systems language. The goal was to implement the full SSB-pub protocol, running on a Raspberry Pi (here, version zero), along with buttons and OLED screen as a way to favor direct material engagement with the kind of hardware that is usually stored in data centers.

Speaking to one of the developers, he mentioned the need to connect to the rest of the world with his choice of programming language, which justified his move from Python to Rust.

"W*e chose Rust because it gives a nice feeling of presence, that I feel I had lost, or even never really had*"

3. Material and social engagements

Two implementations of the SSB protocol:
- **ssb-go** (written in Go by Planetary.social)
- **rustle** (written in Rust by Sunrise Choir)

In order to go about it, the PeachCloud team actually had the choice between two original implementations: a wrapper around the go implementation, or an adaptation of an existing rust implementation.

Both of these implementations also went in different directions.

The go implementation tended towards having large-scale web systems, that was developed by a large scale social network (Planetary)

The rust implementation tended towards “a team working on solarpunk social protocols with Scuttlebutt”, with references to a specific kind of fiction → NEW IMAGINARIES

3. Material and social engagements

Programming as a language community

"*accessible to new developers and maintainable by existing developers*

friendly to other human and computer languages, composed of orthogonal parts that each focus on a separate concern"

Sunrise Choir. (2020). About Sunrise Choir—Open Collective. OpenCollective.Com. https://opencollective.com/sunrise-choir

glyph. (2020). July—October 2020 Update—PeachCloud. OpenCollective.Com. https://opencollective.com/peachcloud/updates/july-october-2020-update

And we can see a quote from Sunrise Choir here: putting natural and machine languages on the same footing.

This aspect of linguistic community of practice is also important to the SSB foundation itself, as it funds tutorials written in Rust, in order to onboard as many people as possible, and make it their own (Iykin, 2021).

So if Rust chosen by other decentralized projects in part due to its community-centered approach (it could also be interesting to consider the genesis of the Rust language at the Mozilla foundation, a pre-existing, community-oriented non-profit foundation, which aims at "empowering everyone", a multiplicitly of local individuals).

3. Material and social engagement

PeachCloud as a hardware project was put on pause, due to language and funding.

The sociality of the language led to a new project.

Problem: the project ended due to a lack of a non-JS implementation (then again assuming a platform/connectivity) → switch to fulltime work on the Rust wrapper, which is a way of showing that programming languages might sometime create path dependencies?

So what we could see here was an attempt to chose a language for two deliberate reasons: in order to engage materially with a infrastructure which felt too remote, and to engage socially with other people writing in the same language.

Reflections

Reflections

The creation of some programming languages facilitate thinking about global scales.

Programming languages can be used to implement ideas of how to do think differently, and then do things efficiently, at a different scale.

The level of abstraction in programming languages can be reconnected to the engagement with the materiality of computing.

So here's what we have said so far...

Particularly, while the last two examples show ways of scaling down from the current state of things, they also suggest interesting intertwinings between ecological sustainability (using less resources), and social sustainability, by reframing spaces of dependence.

Reflections

Scales as “different networks of multilocalities”, materialized in online/offline gradients.

Gibson-Graham, J. K. (2002). Beyond Global vs. Local: Economic Politics Outside the Binary Frame. In Geographies of Power (pp. 25–60). John Wiley & Sons, Ltd. https://doi.org/10.1002/9780470773406.ch1

While the global has been assumed to be more important than the local, we can also reconsider this dichotomy. Rather than global as hypercapitalist or local as anticapitalist, information networks, and the technologies and languages that underpin such networks make such a conception more tangible and more actionable.

“Code is the only language that does what it says”

Reflections

Scales are socio-technical assemblages:

a technical limitation can usher social transformation.

Kogawa, T. (1990). Toward Polymorphous Radio.
http://anarchy.k2.tku.ac.jp/non-japanese/radiorethink.html

Shirky, C. (2004). Shirky: Situated Software.
https://gwern.net/doc/technology/2004-03-30-shirky-situatedsoftware.html

Scaling down a technology also has social effects.

This particular approach can find its echoes in Tetsuo Kogawa's processes of mini-FM, or ultra-local radio broadcasting. While the nature of hertzian waves depends on the electrical power that they are being created with, the implicit impetus for radio broadcasting aims at furthering one's reach as much as possible. Kogawa's project, building on a loophole in Japanese regulation which allowed very weak radio broadcasts, consisted in setting up bloc- or neighborhood-sized broadcast infrastructure, with the unexpected, but welcome side-effect that the listeners could walk to the radio station and say hi to the broadcasters (Kogawa, 1990).

For Kogawa, the concrete localization of technology thus changed the frame of thought from broadcast to individuals, effectively also affecting the cognitive categories of what is considered a valid audience, or an interesting program. Here, the material and the ideal in the production of scale enter in a dialogue through the execution of technological means.

This is something which Clay Shirky has highlighted as situated software: a combination of local scale and social involvement. Between social, local and computing, situated software offers an interesting counterpoint to the tendency of programming languages towards abstraction.

Reflections

Domain-Specific Languages

- topics
- participants
- locations

Fishman, J. A., Cooper, R. L., & Newman, R. M. (1971). Bilingualism in the Barrio. Indiana University.

So this contribution has attempted to start thinking about the notion of scale from computing into geographic terms. There are other terms that we might benefit in examining what the different interpretations are.

Domains in computing is just about the computational domain: what is the digitized data that needs to be manipulated

But if we look at it from a social science perspective, domains have more than just functional practice:

- the location
- the participants
- the topic

It's mostly used in socio-linguistics, but that might be an interesting lens to think about it: whose language are we speaking and under which conditions?

Conclusion

Environments

- test
- dev
- prod
- ... ?

Another term which would be interesting to reconsider is that of “environment”. Traditionally, in software engineering, the environments are either about testing, development or production. Are there different environments in which generic software can run? Low-power environments? Brown-energy environments? Does performance always need to be maximized by default?

Thank you!

pierre@enframed.net