Learning from ALTO-SPCE Code Review

Qiao Xiang
xiangq27@gmail.com

1 ALTO-SPCE Code Review

1.1 What did we do?

1. Jensen led a line-by-line review on the ALTO-SPCE module, including the YANG model and the services provided by the module;
2. Austin led a line-by-line review on the path computation class, elaborating how we implement hop-count-constrained routing and bandwidth-constrained routing;
3. During the review, Cezar, Tony, Qiao and other attendees gave comments and suggestions to ensure the logic correctness of the code, and identified unfinished function of the module;
4. The whole team made a plan to fix found issues based on previous discussions.

1.2 What did we do wrong?

1. We mainly focused on whether the code is logically correct;
2. We overlooked the fact that the current implementation does not support the original API;
3. We only focused on the correctness of very few special cases in the path computation service;
4. We ignored that in practice users may need far more path computation cases to be handled;
5. We underestimate the capability of exhaustive search in enabling generic algorithm, exaggerated its inefficiency of, and try to provide "more efficient" algorithm only for special cases.

1.3 Lessons learned

1. Code review is not just about logical correctness of the code, but also the consistency between design and implementation.
2. Once we have a simple, clean yet efficient API. The major principle during implementation is to wrte algorithm that does not change the API;
3. Exhaustive searching is not always a bad thing in practice, especially when problem scale is relatively small;
4. A generic implementation that enables code reusability is extremely important not only for performance, but for future product update.

2 Guideline for Future Code Review

1. Is the current implementation consistent with the original design? If not, what are the reasons?
2. If the original API is not supported by the current implementation, is it **absolutely** necessary? If not, the implementation **must** be revised to cater the design of API.
3. What use cases can the current implementation cover? Does it provide a generic algorithm, or only covers several special cases? A generic algorithm should be provided in the implementation to ease code reuse and future design/implementation improvement.
4. The code must be logically correct.
5. Sufficient comments should be provided in the code.
6. Consistency and persistence should be ensured.

7. Exceptions must be captured and handled appropriately.
8. Performance must be considered.
9. Simple test case should be provided so that reviewers can run test offline.